Reasoning about Probabilistic Algorithms

Josyula R. Rao*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

May 14, 1990

Abstract

The use of randomization in the design and analysis of algorithms promises simple and efficient algorithms to difficult problems, some of which may not have a deterministic solution. This gain in simplicity, efficiency and solvability results in a tradeoff of the traditional notion of absolute correctness of algorithms for a more quantitative notion: correctness with a probability between 0 and 1. The addition of the notion of parallelism to the already unintuitive idea of randomization makes reasoning about probabilistic parallel programs all the more tortuous and difficult.

In this paper, we address the problem of specifying and deriving properties of probabilistic parallel programs that either hold deterministically or with probability one. We present a proof methodology based on existing proof systems for probabilistic algorithms, the theory of the predicate transformer and the theory of UNITY. Although the proofs of probabilistic programs are slippery at best, we show that such programs can be derived with the same rigor and elegance that we have seen in the derivation of sequential and parallel programs. By applying this methodology to derive probabilistic programs, we hope to develop tools and techniques that would make randomization a useful paradigm in algorithm design.

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658-065 and by the Office of Naval Research, Contract N00014-90-J-1640.

Contents

0 Introduction

0.1 Motivation

Ever since Michael Rabin's seminal paper on Probabilistic Algorithms [Rab76], it has been widely recognized that introducing randomization in the design and analysis of algorithms has several advantages. Often these algorithms are simpler and more efficient — in terms of time, space and communication complexity — than their deterministic[0] counterparts [Rab76; Rab82b; Rab82a; Her89; BGS88]. With the advent of multi-processing and distributed computing, it has been realized that for certain problems, it is possible to construct a probabilistic algorithm where no deterministic one exists. This is true especially for problems involving the resolution of symmetry. In the last decade several such algorithms for synchronization, communication and coordination between parallel programs have appeared in the literature [FR80; IR81; LR81; CLP84; Her89].

This gain in simplicity, efficiency and solvability is not without a price. An integral part of algorithm design is a proof of correctness and for probabilistic algorithms, one has to sacrifice the traditional notion of correctness for a quantitative notion — correctness with a probability between 0 and 1.

In this paper, we address this problem of correctness for probabilistic parallel programs. Since the term *probabilistic* has been used with various connotations, we begin with an informal definition of the class of algorithms of interest. We are interested in those algorithms which in addition to the usual deterministic transitions permit *coin-tossing* as a legal transition. Informally, we allow transitions of the form

$$x \ := \ \text{outcome of the toss of a } \textit{fair} \text{ coin}$$

This simple addition enables us to concisely express many known probabilistic algorithms.

We distinguish between two notions of correctness — *deterministic* and *probabilistic*. The former is defined to capture the traditional notion of absolute correctness of our algorithms. For the latter, rather than specify a quantitative measure, we reason about the more qualitative notion — correctness *with probability 1*. Informally, a property is said to hold with probability 1, if for all schedules, the measure of the set of execution sequences in which the property is attained is 1. In the sequel, properties of interest will be classified as *deterministic* or *holding with probability 1* depending on the notion of correctness used. Note that even probabilistic programs have deterministic properties.

Erring on the side of caution, we require *safety* properties to hold deterministically. For *progress*, we are interested in proving a restricted class of properties, namely those that are attained with probability 1. Furthermore, we are interested in those progress properties that are independent of the actual probability values attached to the individual alternatives

[0]We use the term *deterministic* to mean *non-probabilistic*.

of a probabilistic transition: the only assumption we allow ourselves is that these values are non-zero and the sum of the values over a given transition is 1. Proofs of even such a constrained class of properties can be quite tenuous and tricky[1] : a suggestion for a proof principle for such properties first appeared in [HSP83]. The proof principle was shown to be sound and complete for a class of finite state programs — that is programs with a fixed number of processes and variables ranging over finite domains. The proof principle provided the basis for a decision procedure for mechanically determining whether a program satisfied a progress property with probability 1.

This proof principle has been the essence of several proof systems proposed since then. In [LS82], the authors generalize the temporal propositional logic of linear time to handle random events as well. In [HS84], two propositional probabilistic temporal logics based on temporal logics of branching time are presented. However, they do not address the methodological question of designing probabilistic programs.

In [Pnu83], Pnueli introduced the concept of *extreme fairness*. One of the results of the paper shows that a property that holds for all extremely fair computations would hold with probability 1. Thus, extreme fairness is a sufficient condition to be satisfied by a scheduler in executing probabilistic programs. A proof rule based on extreme fairness and linear time temporal logic was presented. This proof rule has been used to prove some difficult algorithms in [Pnu83; PZ84; PZ86]. Though this method was the first to tackle sizeably complex probabilistic programs, it has been shown to be incomplete. That is, there exist properties expressible in linear time temporal logic that hold with probability 1 but do not hold on some extremely fair computation. Also, the proof rule is so complicated that the authors resort to pictorial representations. Some of the complexity stems from the interplay of randomization and parallelism but their methodology does little to alleviate it. In [Zuc86], Zuck introduced α-fairness and showed that temporal logic properties that hold with probability 1 for a finite state program are precisely those that hold over all α-fair computations of the program.

In [HS85], the proof principle of [HSP83] is generalized to develop conditions for the qualitative analysis of *infinite state* probabilistic programs. To the author's knowledge, no proof system incorporating these conditions has yet been proposed.

[1] For a compelling example of how unintuitive probabilistic reasoning can be, consider the following example: two coins are independently tossed ad infinitum. Is it the case that with probability 1, the system reaches a state in which both coins show heads ? For details, see Example 0 in Section 10.

Researchers have also investigated the question of probabilistic model checking. That is, given a finite state probabilistic program and its temporal logic specification, do the computations of the program satisfy the specification with probability 1 ? This question was first raised in [Var85] and solved by automata theoretic methods extended to incorporate probabilistic transitions. The time complexity of the algorithm suggested to answer this question is double exponential in the size of the specification. In [PZ89], Pnueli and Zuck develop a tableaux-based method to model checking and present an algorithm whose time complexity is single exponential in the size of the specification. This line of research diverges from ours: these methods are essentially a posteriori — they assume that a program is given, whereas we wish to derive the program from the specification.

In [CM88], Chandy and Misra introduce UNITY — a formalism to aid in the specification and derivation of parallel algorithms. Their thesis is that a small theory — a computation model and its associated proof system — is adequate for clearly stating and reasoning about specifications and developing programs for a variety of application areas. Our thesis is that probabilistic parallel algorithms can be specified, refined and derived with the same rigor and elegance that applies to parallel algorithms. By synthesizing ideas from [HSP83; Pnu83], the theory of predicate transformers [DS90] and UNITY [CM88] we construct a theory to reason about probability and parallelism.

0.2 Contributions

We begin by presenting a computational model in which the basic elements of synchrony (modelled as a multiple assignment), asynchrony (modelled as nondeterministic choice) and probabilistic choice are chosen as primitives. Unlike state-based computational models, we do not reason about execution sequences: we choose to reason about properties of programs. In our model, a program is a set of probabilistic multiple assignment statements and an execution of a program proceeds by repeatedly picking a statement from the set and executing it, with the caveat of *unconditional fairness* — that is, in an infinite execution sequence every statement is executed infinitely often. While *unconditional fairness* is required in the selection of a statement to be executed, we require *extreme fairness* in the selection of an alternative of a probabilistic statement.

By defining the *weakest precondition* for a probabilistic statement appropriately, we show that the UNITY relation **unless** and its associated theory can be used to reason about the safety properties of probabilistic programs as well. This is in keeping with our decision to treat all safety properties deterministically. In a like manner, we show how the theory of the UNITY relations **ensures** and $\mapsto$ (read, *leads-to*) can be extended to reason about progress properties that hold *deterministically.*

To specify and verify progress properties that *hold with probability 1*[2], the **wp**-semantics are not adequate. It is necessary to define a new predicate transformer **wpp** (read, *weakest probabilistic precondition*) to capture the inherent non-determinacy of the probabilistic construct. The **wpp** is the dual of **wp** and this is reflected in its properties. It turns out that the predicate transformer **wpp** alongwith the notion of *extreme fairness* provides the right generalization of **wp** (or Hoare triples, for that matter). The predicate **wpp** $.s.X$ characterizes all possible states such that if statement s is executed infinitely often from such a state, then infinitely often the execution of s terminates in a state satisfying predicate X. Furthermore, it provides the basis for generalizing the relations **unless**, **ensures** and $\mapsto$ to new relations **upto**, **entails** and $\models\!\!\Rightarrow$ (read, *probabilistic leads-to*) respectively. Using this small set of operators, we construct a powerful theory in which specifications can be clearly stated and refined using a set of inference rules: furthermore the choice of operators is such that they provide heuristic guidance in extracting the program text from the final specification. We have investigated the properties of the operators in detail.

As with other proof theories, our proof theory is not compositional with respect to general progress properties. However, we have results on deriving basic progress properties of a composite probabilistic program from those of its components. Specifically, **unless**, **upto**, **ensures** and **entails** properties compose in our model.

One of the important features of our theory is that our operators are not powerful enough to reason about the individual state transitions of a program. As a consequence, we are able to avoid the incompleteness of [Pnu83]. We show that our proof system is sound and complete for proving properties of *finite state* probabilistic programs.

Our proof system is novel in that it shows that probabilistic programs are amenable to the same process of specification, refinement and verification as sequential and parallel programs. We illustrate our proof system by examples from random walk and (two process) mutual exclusion problems. Furthermore, our proof system allows both probabilistic and deterministic properties to be manipulated within a unified framework. This allows one to reuse proofs and reason in a compositional manner. The most complicated example that we have proved in our system is the paradigm of *eventual determinism* [Rao90].

[2]To see the necessity of a new operator, see Example 1 in Section 10

0.3 Plan of Paper

After a short introduction to our notation and preliminary theorems in Section 1, we present the format of a deterministic and a probabilistic statement and their **wp**-semantics in Section 2. Several properties of the **wp** of these statements are derived. In Section 3, we use these properties of **wp** to extend the UNITY operator **unless** to reason about safety properties of probabilistic programs. Section 4 summarizes the progress operators of UNITY and shows how the same operators can be used to reason about *deterministic* progress properties of probabilistic programs. Section 5 introduces the predicate transformer **wpp**. We present several theorems relating the **wp** and **wpp** in Section 6. In Section 7, we define operators to reason about progress properties which *hold with probability 1.* Section 8 introduces an induction principle for the $\models\!\!\Rightarrow$. The following section introduces the Substitution Axiom. Section 10 contains a short exposition on program composition. In Section 11, we address soundness and completeness issues for our logic. We illustrate the application of our theory by classic examples from random walks and two process mutual exclusion in Section 12.

1 Notation and Terminology

We will use the following notational conventions: the expression

$$\langle \underline{Q}x \; : \; r.x \; : \; t.x \rangle$$

where $\underline{Q} \in \{\forall, \exists\}$, denotes quantification over all $t.x$ for which x satisfies $r.x$. We call x the **dummy**, $r.x$ the **range** and $t.x$ the **term** of the quantification. We adopt the convention that all formulae are quantified over all free variables occurring in them (these are variables that are neither dummies nor program variables).

Universal quantification over all program variables is denoted by surrounding a predicate by square brackets ([], read: *everywhere*). This unary operator has all the properties of universal quantification over a non-empty range. For a detailed discussion of this notation the reader is referred to [Dij].

For an assignment statement of the form $x := e$, we denote the predicate **wp** $.``x := e".X$ by $\{x :=$

$e\}X$.

To summarize, angle brackets ($\langle\ \rangle$) denote the scope of quantification, square brackets ([ ]) denote quantification over an anonymous state space and curly brackets ({ }) are reserved for the weakest precondition of an assignment.

We write f,g,h, to denote predicate transformers, X,Y,Z to denote predicates on program states and $\mapsto$, $\leadsto$ etc. to denote relations on predicates. For relations **R**, **S** on predicates, we say that **R** is *stronger* than **S** (in formulae $\mathbf{R} \Rightarrow \mathbf{S}$), if and only if $\langle\forall X, Y :: X\,\mathbf{R}\,Y \;\Rightarrow\; X\,\mathbf{S}\,Y\rangle$. For predicate transformers f, g, we say that f is *stronger* than g if and only if $\langle\forall X :: [f.X \Rightarrow g.X]\rangle$.

The other operators we use are summarized below, ordered by increasing binding powers.

$$\begin{gathered} \equiv, \not\equiv \\ \Leftarrow, \Rightarrow \\ \models\!\!\Rightarrow, \mapsto, \leadsto, \text{ etc.} \\ \wedge, \vee \\ \neg \\ =, \neq, \leq, <, \geq, > \\ +, - \\ \text{“.” (function application)} \end{gathered}$$

All boolean and arithmetic operators have their usual meanings.

Next we define a number of junctivity[3] properties for our predicate transformers. The following definitions and theorems have been taken from [DS90].

Definition 0 *A predicate transformer f is said to be* conjunctive *over a bag of predicates V if and only if*

$$[f.\langle\forall X \,:\, X \in V \,:\, X\rangle \equiv \langle\forall X \,:\, X \in V \,:\, f.X\rangle]$$

Definition 1 *A predicate transformer f is said to be* disjunctive *over a bag of predicates V if and only if*

$$[f.\langle\exists X \,:\, X \in V \,:\, X\rangle \equiv \langle\exists X \,:\, X \in V \,:\, f.X\rangle]$$

In other words, the conjunctivity of f describes the extent to which f distributes over universal quantification and its disjunctivity describes how it distributes over existential quantification. The less restricted the V, the stronger the type of junctivity Accordingly, we can distinguish the following types of junctivity:

- *universally junctive* : junctive over all V.
- *positively junctive* : junctive over all non-empty V.
- *denumerably junctive* : junctive over all non-empty V with denumerably many distinct predicates.
- *finitely junctive* : junctive over all non-empty V with a finite number of distinct predicates.
- *and-continuous* : conjunctive over all non-empty V, the distinct predicates of which can be ordered as a monotonic sequence.
- *or-continuous* : disjunctive over all non-empty V, the distinct predicates of which can be ordered as a monotonic sequence.
- *monotonic* : junctive over all non-empty V, the distinct predicates of which can be ordered as a monotonic sequence of finite length.

The various types of junctivity are related by the following theorem.

Theorem 0 *Relating Junctivity Properties :*

- *(universally junctivity $\Rightarrow$ positive junctivity)*
- *(positive junctivity $\Rightarrow$ denumerable junctivity)*
- *(denumerable conjunctivity $\Rightarrow$ finite conjunctivity and and-continuity)*
- *(denumerable disjunctivity $\Rightarrow$ finite disjunctivity and or-continuity)*
- *Both finite conjunctivity and and-continuous $\Rightarrow$ monotonicity*
- *Both finite disjunctivity and or-continuity $\Rightarrow$ monotonicity*

1.1 Proof Format

Most of our proofs will be purely calculational in the sense that they will consist of a number of syntactic transformations instead of semantic reasoning steps. For manipulating formulae of predicate calculus, we use a proof format that was proposed by Feijen, Dijkstra and others, and that greatly facilitates this kind of reasoning.

For instance, a proof that $[A \equiv D]$ could be rendered in our format as

$$\begin{array}{ll} & A \\ = & \{\text{hint why } [A \equiv B]\} \\ & B \\ = & \{\text{hint why } [B \equiv C]\} \\ & C \\ = & \{\text{hint why } [C \equiv D]\} \\ & D \end{array}$$

[3]We use the term *junctive* and its noun form to stand for either *conjunctive* or *disjunctive*.

We also allow other transitive operators in the left-most column. Among these are the more traditional *implies* ($\Rightarrow$) but also for reasons of symmetry, *follows-from* ($\Leftarrow$). For a more thorough treatment of this subject the reader is referred to [DS90].

For manipulating formulae that may depend on several hypotheses, we retain the same calculational style but introduce a different format. Each row of the proof format looks as follows

```
<proof step # >    <formula>
                             ,hint why the formula holds
```

The advantage of this format is that if a particular step depends on several preceding proof steps rather than just the immediate predecessor then this dependence can be explicated in the hint by indicating the step numbers of the hypotheses. This will be particularly useful for proving temporal properties of programs.

2 The Computational Model

Our computational model is the same as that of UNITY. Our programs consist of three parts: a collection of variable declarations, a set of initial conditions and a finite set of statements. As in UNITY, we call these sections **declare**, **initially** and **assign** respectively. A basic transition in our model is a *probabilistic multiple assignment* statement. This is a generalization of UNITY's deterministic multiple assignment statement.

From an operational point of view, an execution of our program starts from any state that satisfies the initial conditions and proceeds by repeatedly selecting any statement from the **assign** set and executing it, with the constraint that in an infinite execution, each statement is picked infinitely often. This is the only notion of fairness - *unconditional fairness* - that is required in the selection of statements.

We now describe the format and *weakest precondition* (**wp**) semantics of a multiple assigment statement as used in UNITY and a probabilistic assignment statement as used in our computational model.

2.1 Deterministic Statements

The only statement allowed in UNITY is the *multiple assignment(MA)*. This can be informally presented as:

$$MA :: x := e$$

In general, x is a list of variables and e is a list of expressions. We restrict an expression to be a well-defined function of the state. Notice that this allows expressions to be defined by cases, provided that there is a unique, well-defined value for the expression in each state. This is required to guarantee that every assignment statement is *deterministic*. The assignment succeeds only if the numbers and types of variables match the corresponding expressions.

Formally, the *weakest precondition* (**wp**) semantics of a conditional multiple assignment is defined as follows.

$$[\mathbf{wp}.MA.X \equiv \{x := e\}X]$$

Theorem 1 *The predicate transformer* $\mathbf{wp}.MA$ *is universally conjunctive.*

To prove the disjunctivity of $\mathbf{wp}.MA$ requires some more groundwork. We have to make use of the fact that the statement is *deterministic*. Using this, we can show that,

Theorem 2 *The predicate transformer* $\mathbf{wp}.MA$ *is universally disjunctive.*

2.2 Probabilistic Statements

The only statement that we allow in our computational model is the *probabilistic assignment statement (PA)*. This can be informally presented as:

$$PA :: x := e.0 \mid e.1 \mid \cdots \mid e.(k-1)$$

As in a multiple assignment statement, x is a list of variables and each $e.i$ $(0 \leq i \wedge i < k)$ is a list of expressions. Again, the only restriction we impose is that the expression be a well-defined function of the state.

A probabilistic assignment is executed as follows. A *fair, k-sided coin* is tossed. The outcome of the coin toss determines the list of expressions $e.i$ to be assigned to the list of variables x. Thus a PA can give rise to one of k different assignments. Each of these possible assignments will be called a *mode* of the PA.

It is important to notice the following points about a probabilistic assignment statement. Firstly, we do not attach a probability value to a mode: we only require that each mode have a non-zero probability of occurrence and that the sum of probabilities over all the modes equal one. Secondly, in the case of only one mode ($k = 1$), a probabilistic statement specializes to a deterministic statement as in UNITY.

We now formalize the notion of fairness required in selecting the mode to be executed (or equivalently, the fairness required in tossing the coin). Let X be a predicate over program variables. An execution, σ, is *extremely-fair with respect to* X if for all probabilistic statements PA, if PA is executed infinitely often from states of σ satisfying X, then every mode of the PA is executed infinitely often from states of σ satisfying X.

An execution is *extremely fair* if it is extremely fair with respect to all first order expressible predicates X.

In [Pnu83], Pnueli established that to prove that a property holds with probability one, it is sufficient to show that it holds over all extremely-fair executions. Thus by assuming that the execution of the probabilistic statements is extremely fair in our computational model, we are assured by Pnueli's result, that all properties hold with probability one.

To recapitulate, our computational model requires two notions of fairness – *unconditional fairness* in the selection of statements to be executed and *extreme fairness* in the execution of a statement.

Formally, the *weakest precondition* (**wp**) semantics of a probabilistic assignment statement is defined as:

$$[\mathbf{wp}.PA.X \equiv \langle \forall i : 0 \leq i \wedge i < k : \{x := e.i\}X\rangle]$$

We now investigate the junctivity properties of this predicate transformer.

Theorem 3 *The predicate transformer* $\mathbf{wp}.PA$ *is universally conjunctive.*

Theorem 4 *The predicate transformer* $\mathbf{wp}.PA$ *is or-continuous.*

Theorem 5 *The predicate transformer* $\mathbf{wp}.PA$ *is* not *finitely disjunctive.*

Corollary 0 *The predicate transformer* $\mathbf{wp}.PA$ *is truth-preserving.*

$$[\mathbf{wp}.PA.true \equiv true]$$

Corollary 1 *(Law of the Excluded Miracle)*

$$[\mathbf{wp}.PA.false \equiv false]$$

3 Reasoning about Safety

In this section, our aim is to define and develop a theory to reason about the safety properties of a probabilistic program. As emphasized in the introduction, we require safety properties to hold deterministically. Since a UNITY program is a special case of a probabilistic program, we would like the relation to be a generalization of UNITY relation for safety, namely, the **unless**. By doing so, we hope to draw on the extensive repertoire of theorems of **unless** that have already been discovered.

In UNITY, the **unless** relation is defined as follows :

$$\frac{\langle \forall s :: [X \wedge \neg Y \Rightarrow \mathbf{wp}.s.(X \vee Y)]\rangle}{(X \ \mathbf{unless}\ Y)}$$

For this definition of **unless** to satisfy the theory of **unless** as developed in UNITY, it is sufficient for the predicate transformer $\mathbf{wp}.s$ to meet the condition of *universal conjunctivity*. That is,

$$[\langle \forall i :: \mathbf{wp}.s.(X.i)\rangle \equiv \mathbf{wp}.s.\langle \forall i :: X.i\rangle]$$

By Theorem 3, the predicate transformer $\mathbf{wp}.PA$ is universally conjunctive and hence we can use the **unless** relation and its theory, as developed in UNITY to reason about the safety properties of probabilistic programs as well.

4 UNITY and Progress: ensures and $\mapsto$

Probabilistic programs could have progress properties that hold deterministically. In this section, we extend the machinery of UNITY to prove deterministic progress properties of probabilistic programs.

Basic progress properties in UNITY are specified using the **ensures** relation. This is defined as

$$\frac{(X \ \mathbf{unless}\ Y) \wedge \langle \exists s :: [X \wedge \neg Y \Rightarrow \mathbf{wp}.s.Y]\rangle}{X \ \mathbf{ensures}\ Y}$$

For this definition of **ensures** to satisfy the theory of **ensures** as developed in UNITY, it is sufficient for the predicate transformer $\mathbf{wp}.s$ to meet the following conditions.

1. Law of the Excluded Miracle

$$[\mathbf{wp}.s.false \equiv false]$$

2. Finite conjunctivity

$$[\mathbf{wp}.s.X \wedge \mathbf{wp}.s.Y \equiv \mathbf{wp}.s.(X \wedge Y)]$$

By Corollary 1, the predicate transformer $\mathbf{wp}.s$ satisfies the Law of the Excluded Miracle for all the statements that we allow in our probabilistic programs. By Theorem 3, it is universally conjunctive and hence is finitely conjunctive as well. Thus we can use the **ensures** relation and its theory as developed in UNITY to reason about **ensures** properties of our programs.

General progress properties in UNITY are defined using the $\mapsto$ (read, *leads to*). The $\mapsto$ relation is defined to be the strongest relation satisfying the following three conditions.

- Base Case :

$$\frac{X \textbf{ ensures } Y}{X \mapsto Y}$$

- Transitivity :

$$\frac{X \mapsto Y,\ Y \mapsto Z}{X \mapsto Z}$$

- Disjunctivity : For an arbitrary set W,

$$\frac{\langle \forall X : X \in W : X \mapsto Y \rangle}{\langle \exists X : X \in W : X \rangle \mapsto Y}$$

The theorems about $\mapsto$ in [CM88] depend on the properties of the **unless**, **ensures** and the definition of $\mapsto$. We have shown that the properties of **unless** and **ensures** continue to hold even with the general probabilistic statements of our computational model. Thus retaining the UNITY definition of $\mapsto$, we can use the theory developed in [CM88] to reason about the *deterministic* progress properties in our computational model.

The theory of **unless** relation is sufficient to reason about safety properties of probabilistic programs. However the notions of **ensures** and $\mapsto$ are inadequate to reason about progress properties that hold with probability 1. They can be used to prove progress properties that have nothing to do with probabilities.That is, there exist programs for which $X \not\mapsto Y$ but X leads-to Y with probability one. This is illustrated by the first example in Section 10.

In the next three sections, we show how each of the **unless**, **ensures** and $\mapsto$ can be generalized to reason effectively about properties that hold with probability 1.

5 The weakest probabilistic precondition

The predicate transformer **wp** allowed us to define safety properties of probabilistic programs. For defining progress properties, it turns out that **wp**$.s$ is too restrictive. Intuitively, **wp**$.PA$ requires *all* modes of the probabilistic statement PA to behave in the same manner, whereas for progress, it is enough if there *exists* a single helpful mode that establishes a desired property. This weaker notion is nicely captured by the predicate transformer **wpp** (read, the *weakest probabilistic precondition*).

The predicate transformer **wpp** is defined as follows

$$[\mathbf{wpp}.PA.X \equiv \langle \exists i : 0 \leq i \wedge i < k : \{x := e.i\}X \rangle]$$

Note that for a deterministic statement s (that is, a statement with a single mode), the **wpp**$.s$ is the same as **wp**$.s$. The only difference between **wp**$.PA$ and **wpp**$.PA$ is in the presence of an existential quantifier in place of a universal one. In this sense, **wpp**$.PA$ is the dual of **wp**$.PA$ and this is reflected in its properties.

Theorem 6 *The predicate transformer* **wpp**$.PA$ *is and-continuous.*

Theorem 7 *The predicate transformer* **wpp**$.PA$ *is not finitely conjunctive.*

Theorem 8 *The predicate transformer* **wpp**$.PA$ *is universally disjunctive.*

Corollary 2 *The predicate transformer* **wpp**$.PA$ *is truth preserving.*

$$[\mathbf{wpp}.s.true \equiv true]$$

Corollary 3 *The predicate transformer* **wpp**$.PA$ *is strict.*

$$[\mathbf{wpp}.s.false \equiv false]$$

Remark: In introducing the notion of the *weakest precondition*, Dijkstra defines **wp**$.s.X$ as characterizing all possible states, such that if s is executed from a state satisfying **wp**$.s.X$, then the execution of s terminates in a state in which X is true.

The predicate transformer **wpp**$.s.X$ considered along with the notion of extreme fairness generalizes this idea. It characterizes all possible states such that if s is executed infinitely often from a state satisfying **wpp**$.s.X$, then infinitely often the execution of s terminates in a state in which X is true. (End of Remark)

6 Relating wp and wpp

In this section, we present theorems relating the predicate transformers **wp** and **wpp**.

Theorem 9 *For all statements s,*

$$[\mathbf{wp}.s.X \Rightarrow \mathbf{wpp}.s.X]$$

Theorem 10 *For all statements s,*

$$[\mathbf{wp}.s.X \wedge \mathbf{wpp}.s.Y \Rightarrow \mathbf{wpp}.s.(X \wedge Y)]$$

Theorem 11 *For all statements s,*

$$[\mathbf{wp}.s.(X \vee Y) \Rightarrow \mathbf{wp}.s.X \vee \mathbf{wpp}.s.Y]$$

7 Reasoning about Progress

In this section, we develop a relation to reason about progress with probability one. The predicate transformer, **wpp** allows us to generalize the UNITY relations **unless** to **upto** and **ensures** to **entails**. We then introduce the $\rightsquigarrow$ (read, *prompts*), as the reflexive, transitive closure of **entails**. These relations provide the basis for defining $\models\!\Longrightarrow$ – the probabilistic analog of the $\mapsto$.

7.1 upto

We begin by generalizing the relation **unless**. Consider the definition of **unless**.

$$\frac{\langle \forall s :: [X \wedge \neg Y \Rightarrow \mathbf{wp}.s.(X \vee Y)]\rangle}{X \textbf{ unless } Y}$$

We use Theorem 11 to weaken this definition to obtain the definition of **upto**.

$$\frac{\langle \forall s :: [X \wedge \neg Y \Rightarrow \mathbf{wp}.s.X \vee \mathbf{wpp}.s.Y]\rangle}{X \textbf{ upto } Y}$$

Intuitively, X **upto** Y captures the following idea : If X holds at any point during the execution of a program, then either

1. Y never holds and X continues to hold forever, or
2. Y holds eventually (it may hold initially when X holds) and X continues to hold until Y holds, or
3. X continues to hold until $\neg X$ holds eventually; the transition from X to $\neg X$ being made by a statement that *could have* taken it to a state satisfying Y.

The interesting (third) case arises when a probabilistic statement PA is executed in a state satisfying $X \wedge \neg Y$. Suppose not all modes of the PA when executed lead to a state satisfying X and furthermore there exists a mode which will take it to a state satisfying Y. Since there are no guarantees on which mode will be executed, execution of PA can lead to a state satisfying $\neg X$, even though there exists a mode that can take it to Y.

One of the consequences of this definition is that in general **upto** includes **unless** and if all statements are deterministic (i.e. multiple assignments) the definition of **upto** reduces to **unless**.

Theorem 12 *The* **upto** *is a generalization of* **unless**.

$$(X \textbf{ unless } Y) \Rightarrow (X \textbf{ upto } Y)$$

Furthermore for a program consisting of only deterministic statements,

$$(X \textbf{ unless } Y) \equiv (X \textbf{ upto } Y)$$

The relation **upto** is weaker than **unless** and accordingly it enjoys a smaller set of properties.

1. Reflexivity and Anti-Reflexivity :

$$X \textbf{ upto } X$$

$$X \textbf{ upto } \neg X$$

2. Consequence Weakening :

$$\frac{X \textbf{ upto } Y, Y \Rightarrow Z}{X \textbf{ upto } Z}$$

3. Partial Conjunction :

$$\frac{X \textbf{ upto } Y, \quad X' \textbf{ upto } Y'}{(X \wedge X') \textbf{ upto } ((X' \wedge Y) \vee Y')}$$

4. Simple Conjunction and Simple Disjunction :

$$\frac{X \textbf{ upto } Y \quad X' \textbf{ upto } Y'}{\begin{array}{ll}(X \wedge X') \textbf{ upto } (Y \vee Y') & \text{, simple conjunction}\\ (X \vee X') \textbf{ upto } (Y \vee Y') & \text{, simple disjunction}\end{array}}$$

5. Conjunction with **unless** :

$$\frac{X \textbf{ unless } Y \quad X' \textbf{ upto } Y'}{(X \wedge X') \textbf{ upto } (X \wedge Y') \vee (X' \wedge Y) \vee (Y \wedge Y')}$$

Many of the properties of **unless** are not inherited by **upto**. In particular, conjunction, disjunction, general conjunction and general disjunction [CM88] do not hold for **upto**. Furthermore the rule of cancellation

$$\frac{X \textbf{ upto } Y, \quad Y \textbf{ upto } Z}{(X \vee Y) \textbf{ upto } Z}$$

does not hold for **upto**. This is not a problem as **upto** is almost never used for specifications; its utility lies in defining operators for progress. There will be few manipulations involving **upto**.

7.2 entails

We propose a new relation **entails** to generalize **ensures**. Consider the definition of **ensures**.

$$\frac{(X \textbf{ unless } Y) \wedge \langle \exists s :: [X \wedge \neg Y \Rightarrow \textbf{wp}.s.Y] \rangle}{X \textbf{ ensures } Y}$$

We use Theorem 9 to weaken this definition to obtain the definition of **entails**.

$$\frac{(X \textbf{ upto } Y) \wedge \langle \exists s :: [X \wedge \neg Y \Rightarrow \textbf{wpp}.s.Y] \rangle}{X \textbf{ entails } Y}$$

The intuitive meaning of $(X$ **entails** $Y)$ is that if X is infinitely often true in a computation, then Y is infinitely often true. The claim that Y is infinitely often true is justified as follows. Let an X-state be a state satisfying predicate X. Suppose $X \wedge \neg Y$ holds at some point in the execution of the program. By the first conjunct the only way a program can reach a $\neg X$-state, is to execute a statement that *may lead to* a Y-state. Note that the second conjunct assures us of the existence of such a statement s which has a mode, whose execution in a $(X \wedge \neg Y)$-state, would lead to a Y-state. By *unconditional fairness*, s must be executed, causing the program to transit to a $\neg X$-state. If X is infinitely often true then each time the transition from an X-state to a $\neg X$-state is made, it is done by executing a statement whose execution could have resulted in a Y-state. From the finiteness of the set of statements, some statement t whose execution could have lead to a Y-state is executed infinitely often from X-states. By extreme fairness, every mode of t is executed infinitely often from X-states. In particular, the mode leading to a Y-state is executed infinitely often. It follows that Y is infinitely often true.

The ideas introduced in our computational model - unconditional fairness and extreme fairness - were all intended to justify this definition of the **entails** relation. The relation **entails** plays an important role in the design of probabilistic programs. Besides being the keystone of the proof theory of progress properties, it has a methodological significance as well. In extracting a program from a specification, each **entails** property can usually be translated to a single probabilistic statement. This will be illustrated by an example in a later section.

Theorem 13 *The* **entails** *generalizes* **ensures**

$$(X \textbf{ ensures } Y) \Rightarrow (X \textbf{ entails } Y)$$

Furthermore for a program consisting only of deterministic statements,

$$(X \textbf{ ensures } Y) \equiv (X \textbf{ entails } Y)$$

Since **entails** is a generalization of **ensures**, it enjoys a smaller set of properties.

1. Reflexivity :

$$X \textbf{ entails } X$$

2. Consequence Weakening :

$$\frac{X \textbf{ entails } Y,\ Y \Rightarrow Z}{X \textbf{ entails } Z}$$

3. Impossibility :

$$\frac{X \textbf{ entails } \mathit{false}}{\neg X}$$

4. Conjunction with **unless** :

$$\frac{\begin{array}{c}X \textbf{ entails } Y\\ X' \textbf{ unless } Y'\end{array}}{(X \wedge X') \textbf{ entails } (X \wedge Y') \vee (X' \wedge Y) \vee (Y \wedge Y')}$$

5. Conjunction with **upto** :

$$\frac{\begin{array}{c}X \textbf{ entails } Y\\ X' \textbf{ upto } Y'\end{array}}{(X \wedge X') \textbf{ entails } (X' \wedge Y) \vee Y'}$$

6. Disjunction :

$$\frac{X \textbf{ entails } Y}{(X \vee Z) \textbf{ entails } (Y \vee Z)}$$

Of all the properties of **ensures**, the conjunction rule [CM88], does not hold for **entails**.

7.3 The relation $\rightsquigarrow$

The relation **entails** is tied closely to the program. We abstract from this by defining the relation $\rightsquigarrow$ (read, *prompts*) to be the reflexive, transitive closure of **entails**.

- Base Case :

$$\frac{(X \textbf{ entails } Y)}{(X \rightsquigarrow Y)}$$

- Transitivity :

$$\frac{X \rightsquigarrow Y,\ Y \rightsquigarrow Z}{X \rightsquigarrow Z}$$

The definition of $\rightsquigarrow$ satisfies the following properties.

1. Implication :
$$\frac{X \Rightarrow Y}{X \rightsquigarrow Y}$$

2. Impossibility :
$$\frac{X \rightsquigarrow false}{\neg X}$$

3. Disjunction :
$$\frac{(X \rightsquigarrow Y)}{(X \vee Z) \rightsquigarrow (Y \vee Z)}$$

4. Finite Disjunction :
$$\frac{(X \rightsquigarrow Z), (Y \rightsquigarrow Z)}{(X \vee Y) \rightsquigarrow Z}$$

5. Cancellation :
$$\frac{U \rightsquigarrow V \vee W,\ W \rightsquigarrow X}{U \rightsquigarrow V \vee X}$$

6. PSP (Progress-Safety-Progress) :
$$\frac{X \rightsquigarrow Y,\ U \textbf{ unless } V}{(X \wedge U) \rightsquigarrow (Y \wedge U) \vee V}$$

7. Completion :
For any finite set of predicates, $X.i$, $Y.i$, $0 \leq i < N$:
$$\frac{\begin{array}{c}\langle \forall i :: X.i \rightsquigarrow Y.i \vee Z\rangle \\ \langle \forall i :: Y.i \textbf{ unless } Z\rangle\end{array}}{\langle \wedge i :: X.i\rangle \rightsquigarrow \langle \wedge i :: Y.i\rangle \vee Z}$$

7.4 Probabilistic Leads–to: $\models\!\!\Rightarrow$

In this paper, we shall express all probabilistic progress properties using the $\models\!\!\Rightarrow$ (read, *probabilistic leads-to*). A program has the property $X \models\!\!\Rightarrow Y$ if once X becomes true, Y will become true with probability one. The $\models\!\!\Rightarrow$ is defined to be the strongest relation satisfying the following three axioms.

- Base Case :
$$\frac{X \textbf{ unless } Y,\ X \rightsquigarrow Y}{X \models\!\!\Rightarrow Y}$$

- Transitivity :
$$\frac{X \models\!\!\Rightarrow Y,\ Y \models\!\!\Rightarrow Z}{X \models\!\!\Rightarrow Z}$$

- Disjunctivity : For an arbitrary set W,
$$\frac{\langle \forall X : X \in W : X \models\!\!\Rightarrow Z\rangle}{\langle \exists X : X \in W : X\rangle \models\!\!\Rightarrow Z}$$

According to the first axiom, if X is true at any point in the execution of a program, by X **unless** Y it remains true indefinitely or until Y becomes true. In the former case, X is infinitely often true and by $X \rightsquigarrow Y$, Y is infinitely often true. In either case, Y becomes true. The second axiom ensures that $\models\!\!\Rightarrow$ is transitively closed and the third axiom ensures that $\models\!\!\Rightarrow$ is disjunctively closed.

Probabilistic leads-to is a generalization of the UNITY leads-to. That is,

Theorem 14 $(X \mapsto Y) \Rightarrow (X \models\!\!\Rightarrow Y)$

The probabilistic leads-to ($\models\!\!\Rightarrow$) enjoys many of the properties of $\mapsto$.

1. Implication :
$$\frac{X \Rightarrow Y}{X \models\!\!\Rightarrow Y}$$

2. Impossibility :
$$\frac{X \models\!\!\Rightarrow false}{\neg X}$$

3. General Disjunction :
$$\frac{\langle \forall m : m \in W : X.m \models\!\!\Rightarrow Y.m\rangle}{\langle \exists m : m \in W : X.m\rangle \models\!\!\Rightarrow \langle \exists m : m \in W : Y.m\rangle}$$

4. Cancellation :
$$\frac{U \models\!\!\Rightarrow V \vee W,\ W \models\!\!\Rightarrow X}{U \models\!\!\Rightarrow V \vee X}$$

5. PSP (Progress-Safety-Progress) :
$$\frac{X \models\!\!\Rightarrow Y,\ U \textbf{ unless } V}{(X \wedge U) \models\!\!\Rightarrow (Y \wedge U) \vee V}$$

6. Completion :
For any finite set of predicates, $X.i$, $Y.i$, $0 \leq i < N$:
$$\frac{\begin{array}{c}\langle \forall i :: X.i \models\!\!\Rightarrow Y.i \vee Z\rangle \\ \langle \forall i :: Y.i \textbf{ unless } Z\rangle\end{array}}{\langle \wedge i :: X.i\rangle \models\!\!\Rightarrow \langle \wedge i :: Y.i\rangle \vee Z}$$

8 An Induction Principle for Probabilistic Leads-to

An interesting property of $\models\!\!\Rightarrow$ is its transitivity; and one can use this to formulate a principle of induction over well-founded sets.
Let $(W, \prec)$ be a well-founded set. Let M be a metric mapping the states of the program to W. Then the induction principle states

$$\frac{\langle \forall m : m \in W : X \wedge (M = m) \models\!\!\Rightarrow (X \wedge M \prec m) \vee Y \rangle}{X \models\!\!\Rightarrow Y}$$

Intuitively, the hypothesis of the induction principle states that from any program state in which X holds, execution of the program leads to a state in which either Y is true or the value of the metric is decreased. Since the range of the metric is a well-founded set, the metric cannot decrease indefinitely. It follows that a state satisfying Y is attained.

9 Substitution Axiom

The substitution axiom has been introduced in UNITY as a generalization of Leibniz's principle of substitution of equals for equals. Informally, if $x = y$, then x can be substituted for y in all program properties. Of particular importance is the case where a program invariant I can be substituted by *true* and vice versa. This is because $[I \equiv true]$.

The importance of the substitution axiom is twofold. Firstly, it is necessary for the completeness of our proof system. Secondly, it allows us to abbreviate cumbersome notation.

10 On program composition

We use the same notions of program composition as UNITY, namely, *union* and *superposition*.

10.1 Composition by union

The *union* of two programs is the union of the sets of statements in the **assign** sections of the two programs. The union of programs F and G is written as $F \; [\!] \; G$. Like set union, it is a symmetric and associative operator. We assume that there are no inconsistencies in the declarations and initializations of the variables in the two programs.
The study of program composition by union is facilitated by the the union theorem. An important condition on all cases of the theorem is that any application of the substitution axiom in the proof of a property of either the component or the composite program can only use an invariant of the *composite* program.

Theorem 15 Union Theorem:

- $(X$ **unless** Y **in** $F \wedge X$ **unless** Y **in** $G) \equiv (X$ **unless** Y **in** $F \; [\!] \; G)$
- $(X$ **ensures** Y **in** $F \wedge X$ **unless** Y **in** $G) \vee (X$ **unless** Y **in** $F \wedge X$ **ensures** Y **in** $G) \equiv (X$ **ensures** Y **in** $F \; [\!] \; G)$
- $(X$ **upto** Y **in** $F \wedge X$ **upto** Y **in** $G) \equiv (X$ **upto** Y **in** $F \; [\!] \; G)$
- $(X$ **entails** Y **in** $F \wedge X$ **upto** Y **in** $G) \vee (X$ **upto** Y **in** $F \wedge X$ **entails** Y **in** $G) \equiv (X$ **entails** Y **in** $F \; [\!] \; G)$
- $(X$ **entails** Y **in** $F \wedge X$ **unless** Y **in** $G) \vee (X$ **unless** Y **in** $F \wedge X$ **entails** Y **in** $G) \Rightarrow (X$ **entails** Y **in** $F \; [\!] \; G)$

10.2 Conditional Properties

The union theorem illustrates that basic progress properties compose, that is, the property holds of the composite program if its holds of the components. This is not the case with $\leadsto$, $\mapsto$ and $\models\!\!\Rightarrow$.

To address this shortcoming, we resort to *conditional* properties as in UNITY. All program properties seen thusfar have been expressed using one or more relations – **unless**, **ensures**, **upto**, **entails**; these properties are called *unconditional* properties. A conditional property has two parts – *a hypothesis* and *a conclusion*, each of which is a set of unconditional properties. Both the hypothesis and the conclusion can be properties of the F, G or $F \; [\!] \; G$, where G is a generic program. The meaning of a conditional property is as follows : Given the hypothesis as a permise, the conclusion can be proven from the text or specification of F. Thus in proving properties, a conditional property is used as an inference rule. The interested reader is referred to [CM88] for further elucidation.

10.3 Superposition

The second structuring operator that we employ in our proofs is the *superposition* operator. This is exactly the same operator as in UNITY. We recapitulate the salient details.

Unlike program union, program superposition is an *asymmetric* operator. Given an *underlying program* (whose variables will be called *underlying variables*), superposition allows it to be transformed by the application of the following two rules.

1. Augmentation Rule. A statement s in the underlying program may be transformed to the statement $s \| r$ where r is a statement that does not assign to the underlying variables and is executed *in synchrony* with s.
2. Restricted Union Rule. A statement r may be added to the underlying program provided that r does not assign to the underlying variables.

By adhering to the discipline of superposition, it is ensured that every property of the underlying program is a property of the transformed program. This is also called the superposition theorem.

11 Comments on Soundness and Completeness

In [Pnu83], Pnueli presents a theorem that shows that every property that holds of all extremely fair executions holds with probability 1 for the program. In our computational model, all executions are extremely fair. Any property that we prove in our computational model, holds for all such executions and thus holds with probability 1.

In this section, we informally argue that our logic is complete for finite state programs. By the results of [HSP83], if a property that expresses reachability holds with probability 1 for a finite state program, it induces a decomposition on the (finite) set of states that can be visited before the goal states are reached. Each partition of this decomposition satisfies certain conditions. These conditions guarantee that in any unconditionally fair and extremely fair computation the property is attained. This shows that our model is sound and complete for proving properties that hold with probability 1 of finite state programs.

Let Σ be the set of states of the program. Let s be an initial state and let $X (X \subset \Sigma)$ be the set of final states of the program, with $s \notin X$. Define $\hat{I}$ as the set of all states that can be reached (with a non-zero probability) from s before a state in X is reached, using any finite sequence of processes. $\hat{I}$ includes s and is disjoint from X. Furthermore, let K be the set of processes of the program and let $P^k_{i,J}$ be the probability of process k taking the system from state i to any state in set J. One of the main results of [HSP83] is that assuming s, X, Σ and $\hat{I}$ as above and assuming $\hat{I}$ is *finite*, the following two conditions are equivalent.

- $\models s \models\!\!\Rightarrow X$
- There exists a decomposition of $\hat{I}$ into disjoint sets $I_1, I_2, \ldots, I_m$ such that, if we put $J_m = \cup_{r=0}^{m} I_r$, $m = 0, 1, \ldots, n$, with $I_0 = X$, then for each $m = 1, 2, \ldots, n$ we have the following:
  - For each $i \in I_m$, $k \in K$, if $P^k_{i,J_{m-1}} = 0$, then $P^k_{i,I_m} = 1$.
  - There exists $k \equiv k(m) \in K$ such that, for each $i \in I_m$, $P^k_{i,J_{m-1}} > 0$

The first part of the second condition says that if process k can transfer the system from a state in I_m to a state outside I_m, then some k-transitions (with non-zero probability) move the system "down" the chain $\{I_r\}$, towards the goal I_0; the second part ensures the existence of at least one process that would do this for all states in I_m.

Thus given that some progress property holds in a model with probability one, we are guaranteed that the chain $\{I_r\}$ exists. Clearly, $\hat{I}$ **unless** I_0 holds by the definition of $\hat{I}$ and **unless**. For each element of the chain, we can show I_r **entails** J_{r-1}. By using transitivity of $\leadsto$ we can show, $I_r \leadsto I_0$. Using finite disjunction property of $\leadsto$, one can conclude that $\hat{I} \leadsto I_0$. The proof follows from the **unless** property, the $\leadsto$ property and the definition of $\models\!\!\Rightarrow$.

12 Examples

Note that in all examples if a common boolean guard applies to all the modes of a probabilistic assignment, it has been pulled out and written once. This has been done in the interests of brevity.

Example 0: (An Unintuitive Example)

To show how unintuitive, reasoning about probabilistic algorithms can be, consider the following program.

```
declare     x, y : (heads, tails)
assign      x := heads | tails
         [] y := heads | tails
end
```

It can be shown that the property

$$true \models\!\!\Rightarrow (x = heads) \wedge (y = heads)$$

does not hold for the given program. This is because it is possible for the execution of the program to be unconditionally fair with respect to the selection of the coin to be tossed and extremely fair in the tossing of the coins, without reaching a state in which both coins turn up *heads*. Abbreviating *heads* by H and *tails* by T, consider the following segment σ of state transformations: (the state is denoted by the ordered pair giving the values of x and y).

$$(H,T) \xrightarrow{x:=heads} (H,T) \xrightarrow{x:=tails} (T,T) \xrightarrow{y:=tails}$$

$$(T,T) \xrightarrow{y:=heads} (T,H) \xrightarrow{y:=heads}$$

$$(T,H) \xrightarrow{y:=tails} (T,T) \xrightarrow{x:=tails} (T,T) \xrightarrow{x:=heads} (H,T)$$

The sequence σ iterated indefinitely gives an execution sequence which is *unconditionally fair* and *extremely fair*. One way of ensuring that a state satisfying $[(x = heads) \wedge (y = heads)]$ is reached is to use extreme fairness in the scheduling of the statements, rather than unconditional fairness, as illustrated by the program below. This also illustrates the power of extreme fairness over unconditional fairness.

```
declare    $x, y : (H, T)$
assign     $x, y := H,H \mid H,T \mid T,H \mid T,T$
end
```

(End of Example)

Example 1: (From [Pnu83])

Consider the UNITY program :

```
declare      $b : integer$
initially    $b = 0$
assign       $b := b + 1$  if  $(b\,mod\,3) \leq 1$
          ▯  $b := b + 2$  if  $(b\,mod\,3) \leq 1$
end
```

For this program, it is not the case that

$$true \mapsto (b\,mod\,3\ =\ 2)$$

Consider the execution sequence in which the two statements are alternately executed, leading to the following sequence of values for b:

$$0, 1, 3, 4, 6, 7, \ldots$$

This execution sequence is *unconditionally fair* with respect to the two statements but no state of the execution satisfies $(b\,mod\,3\ =\ 2)$. Thus the program does not satisfy the progress property *deterministically*.

Now consider the probabilistic program.

```
declare      $b : integer$
initially    $b = 0$
assign       $b := b + 1 \mid b + 2$  if  $(b\,mod\,3) \leq 1$
end
```

We show that the required property is achieved with probability one, that is

$$true \models\!\!\Rightarrow (b\,mod\,3\ =\ 2)$$

By applying the definition of **wpp** $.s$ it can be shown that **wpp** $.s.((b\,mod\,3) = 2)]$ evaluates to true. Thus

$\langle \exists s :: [true \wedge \neg(b\,mod\,3 = 2)$
$\qquad\qquad \Rightarrow$ **wpp** $.s.((b\,mod\,3) = 2)]\rangle$
$=$ {predicate calculus}
$[\neg(b\,mod\,3 = 2) \Rightarrow true]$
$=$ {predicate calculus}
$true$

0. $\langle \exists s :: [true \wedge \neg(b\,mod\,3 = 2) \Rightarrow$ **wpp** $.s.(b\,mod\,3 = 2)]\rangle$
   ,From above
1. $true$ **upto** $(b\,mod\,3 = 2)$
   ,Tautology for **upto**
2. $true$ **entails** $(b\,mod\,3 = 2)$
   ,From 0, 1 and the definition of **entails**
3. $true$ **unless** $(b\,mod\,3 = 2)$
   ,Tautology for **unless**
4. $true \models\!\!\Rightarrow (b\,mod\,3 = 2)$
   ,From 2,3 and the definition of $\models\!\!\Rightarrow$

(End of Example)

Example 2: (Random walk[4] problems)

At any instant of time a particle inhabits one of the integer points of the real line. At time 0, it starts at the specified point and at each subsequent "clock-tick", it moves from its current position to the new position according to the following rule: with probability p it moves one step to the right and with probability $q = 1 - p$, it moves one step to the left; the moves are independent of each other.

For the random walk problem with no barriers on the real line, it is possible to show that the particle returns to 0 with probability one only if $p = q$. This is also called the *symmetric random walk problem*. Although this property holds with probability one, it is not possible to prove it in our proof system. This is because the property *depends* on the values of the probabilities of the transition, i.e. $p = q$.

There are a class of random walk problems whose progress properties are independent of the values of the probabilities of the transition. As our first example, we consider random walk with two *absorbing* barriers at 0 and M. This means that that the instant, the particle reaches a barrier it is trapped. The

[4]In general, random walks can be in many dimensions and the step size can be arbitrary. For ease of exposition we restrict ourselves to one dimension and a step size of 1.

movement of the particle is modelled by the following program.

```
declare   $x : [0 \ldots M]$
assign    $x := x - 1 \mid x + 1$  if  $(0 < x \wedge x < M)$
end
```

For this program we prove that

$$true \Longmapsto (x = 0) \vee (x = M)$$

We assume, without proof, that

$$\textbf{invariant } (0 \leq x) \wedge (x \leq M)$$

Assume that the range of k is given by $0 < k \wedge k < M$.

0. $\langle \forall k :: (x = k) \textbf{ entails } (x = k - 1)\rangle$
   ,From the program text
1. $\langle \forall k :: (x = k) \leadsto (x = 0)\rangle$
   ,Transitivity of $\leadsto$
2. $\langle \exists k :: (x = k)\rangle \leadsto (x = 0)$
   ,Finite disjunction for $\leadsto$
3. $\langle \exists k :: (x = k)\rangle \leadsto (x = M)$
   ,Proof similiar to 2
4. $\langle \exists k :: (x = k)\rangle \leadsto (x = 0) \vee (x = M)$
   ,Finite Disjunction using 2 and 3
5. $(x = 0) \vee (x = M) \leadsto (x = 0) \vee (x = M)$
   ,Implication for $\leadsto$
6. $\langle \exists k :: (x = k)\rangle \vee (x = 0) \vee (x = M)$
   $\leadsto (x = 0) \vee (x = M)$
   ,Disjunction of 4 and 5
7. $true \leadsto (x = 0) \vee (x = M)$
   ,predicate calculus and substitution axiom
   ,using invariant above
8. $true$ **unless** $(x = 0) \vee (x = M)$
   ,Tautology for **unless**
9. $true \Longmapsto (x = 0) \vee (x = M)$
   ,From 7, 8 and the definition of $\Longmapsto$

As our second example illustrating random walk, consider two *reflecting* barriers to be placed at 0 and M. This means that when the particle reaches the barrier at 0 (or M) it bounces back to 1 (or $M - 1$) with probability one. The movement of the particle is modelled by the following program.

```
declare     $x : [0 \ldots M]$
assign      $x := x - 1 \mid x + 1$  if  $(0 < x \wedge x < M)$
         []  $x := 1$  if  $(x = 0)$
         []  $x := M - 1$  if  $(x = M)$
end
```

For this program, it is easy to show that

$$\textbf{invariant } (0 \leq x) \wedge (x \leq M)$$

The range of k is assumed to be $0 \leq k \wedge k \leq M$. We show that

$$true \Longmapsto (x = 0)$$

0. $\langle \forall k :: (x = k) \textbf{ entails } (x = k - 1)\rangle$
   ,From program text
1. $\langle \forall k :: (x = k) \leadsto (x = 0)\rangle$
   ,Transitivity of $\leadsto$
2. $\langle \exists k :: (x = k)\rangle \leadsto (x = 0)$
   ,Finite Disjunction of $\leadsto$
3. $true \leadsto (x = 0)$
   ,predicate calculus and substitution axiom
   ,using the invariant above
4. $true$ **unless** $(x = 0)$
   ,Tautology for **unless**
5. $true \Longmapsto (x = 0)$
   , From 3,4 and the definition of $\Longmapsto$

As our third example we consider the case of an *absorbing* barrier at 0 and a *reflecting* barrier at M. The movement of the particle would be modelled by the following program.

```
declare     $x : [0 \ldots M]$
assign      $x := x - 1 \mid x + 1$  if  $(0 < x \wedge x < M)$
         []  $x := M - 1$  if  $(x = M)$
end
```

For this program, we assume, without proof, that

$$\textbf{invariant } (0 \leq x) \wedge (x \leq M)$$

The range of k is assumed to be $0 < k \wedge k \leq M$. We show that

$$true \Longmapsto (x = 0)$$

0. $\langle \forall k :: (x = k) \textbf{ entails } (x = k - 1)\rangle$
   ,From program text
1. $\langle \forall k :: (x = k) \leadsto (x = 0)\rangle$
   ,Transitivity of $\leadsto$
2. $\langle \exists k :: (x = k)\rangle \leadsto (x = 0)$
   ,Finite Disjunction of $\leadsto$
3. $(x = 0) \leadsto (x = 0)$
   ,Implication Rule of $\leadsto$
4. $\langle \exists k :: (x = k)\rangle \vee (x = 0) \leadsto (x = 0)$
   ,Finite Disjunction using 2 and 3
5. $true \leadsto (x = 0)$
   ,predicate calculus and substitution axiom
   ,using the invariant above
6. $true$ **unless** $(x = 0)$
   ,Tautology for **unless**
7. $true \Longmapsto (x = 0)$
   , From 5,6 and the definition of $\Longmapsto$

(End of Example)

Example 3: (Two process mutual exclusion)

In this example, we give a brief overview of specification refinement. The example is designed to give a flavor of proof machinery at work.

Specifically, we consider the problem of mutual exclusion between two processes — u, v. Each process u has a variable $u.dine$, which can take one of three values t, h or e, corresponding to *thinking*, *hungry* or *eating*. We abbreviate by $u.t$, $u.h$ and $u.e$, the expressions $u.dine = t$, $u.dine = h$ and $u.dine = e$ respectively. We assume that every thinking process eventually becomes hungry. A hungry process remains hungry till it eats. An eating process eats for a finite time and then transits to thinking.

Specification of Program *mutex*:

(0a) **invariant** $\neg(u.e \wedge v.e \wedge u \neq v)$
(0b) $u.h \models\!\!\!\Rightarrow u.e$

Properties of *mutex*

(D0) $u.h$ **unless** $u.e$
(D1) $u.t \leadsto u.h$

Our first refinement will consist of the introduction of a two-sided coin that can take on values u and v such that if the process u is eating then the coin has the value u

Specification 1 :

(1a) **invariant** $(u.e \Rightarrow coin = u)$
(1b) $u.h \models\!\!\!\Rightarrow u.e$

It is easy to show that (1a) $\Rightarrow$ (0a). Our next refinement is to ensure that the progress property is met. We propose that if a process u is hungry and if the *coin* has value u then the process be allowed to enter the critical section. To avoid starvation, we need to ensure that the coin eventually takes the value of every process. Thus our next specification reads:

Specification 2 :

(2a) **invariant** $(u.e \Rightarrow coin = u)$
(2b) $u.h \wedge (coin = u) \models\!\!\!\Rightarrow u.e$
(2c) $coin = u \models\!\!\!\Rightarrow coin = v$

We show that specification 2 implies specification 1.

0. $(coin = v) \models\!\!\!\Rightarrow (coin = u)$
 ,From 2c
1. $u.h$ **unless** $u.e$
 ,Property D0 of *mutex*
2. $(u.h \wedge coin = v) \models\!\!\!\Rightarrow (u.h \wedge coin = u) \vee u.e$
 ,PSP Theorem on 0 and 1
3. $(u.h \wedge coin = v) \models\!\!\!\Rightarrow u.e$
 ,Cancellation on 2 and 2b
4. $u.h \models\!\!\!\Rightarrow u.e$
 ,Disjunction on 3 and 2b

To implement the progress properties of Specification 2, we propose the following refinement.

Specification 3 :

(3a) **invariant** $(u.e \Rightarrow coin = u)$
(3b) $u.h \wedge (coin = u)$ **ensures** $u.e$
(3c) $u.e$ **entails** $coin = v$

We show that specification 3 implies specification 2.

0. $(coin = u)$ **unless** $(coin = v)$
 ,Tautology for **unless**
1. $u.t \leadsto u.h$
 ,Property D1 and 3a
2. $(u.t \wedge coin = u) \leadsto (u.h \wedge coin = u) \vee (coin = v)$
 ,PSP on 0 and 1
3. $u.h \wedge (coin = u)$ **entails** $u.e$
 ,Generalizing **ensures** in 3b to **entails**
4. $u.e \wedge (coin = u)$ **entails** $(coin = v)$
 ,From 3a and 3c
5. $(coin = u) \leadsto (coin = v)$
 ,Properties of $\leadsto$ on 2, 3, 4
6. $(coin = u) \models\!\!\!\Rightarrow (coin = v)$
 ,From 0, 5 and definition of $\models\!\!\!\Rightarrow$

The final specification suggests the following program.

```
declare coin : $(u, v)$
initially $u.dine,\ v.dine := t,\ t$
assign
    $\langle [\!]\ x : x \in (u, v) :$
            $u.dine := h$  if  $u.t$
            $u.dine := e$  if  $u.h \wedge (coin = u)$
            $u.dine, coin := t, u \mid t, v$  if  $u.e$
    $\rangle$
end
```

(End of Example)

13 Acknowledgement

I am grateful to Professor Jayadev Misra for advising and supporting me. Thanks are also due to Charanjit Jutla and Sumit Ganguly for suggestions and criticisms.

References

[BGS88] Shaji Bhaskar, Rajive Gupta, and Scott Smolka. Probabilistic algorithms: A survey. Private Communication, 1988.

[CLP84] S. Cohen, Daniel Lehmann, and Amir Pnueli. Symmetric and economic solution to the mutual exclusion problem in distributed systems. *Theoretical Computer Science*, 34:215–226, 1984.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Dij] Edsger W. Dijkstra. On structures. EWD 928.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[FR80] Nissim Francez and M. Rodeh. A distributed data type implemented by a probabilistic communication scheme. In *Proceedings of the 21st Symposium on the Foundations of Computer Science*, pages 373–379, 1980.

[Her89] Ted Herman. Probabilistic self-stabilization. Private Communication, 1989.

[HS84] Sergiu Hart and Micha Sharir. Probabilistic propositional temporal logics. In *Proceedings of the 16th Symposium on Theory of Computing*, pages 1–13, 1984.

[HS85] Sergiu Hart and Micha Sharir. Concurrent probabilistic programs, or: How to schedule if you must. *Siam Journal of Computing*, 14:991–1012, 1985.

[HSP83] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.

[IR81] A. Itai and M. Rodeh. The lord of the ring or probabilistic methods for breaking symmetry in distributive networks. Technical Report RJ 3110, IBM, San Jose, 1981.

[LR81] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, Williamsburg, VA, 1981.

[LS82] Daniel Lehmann and S. Shelah. Reasoning with time and chance. *Information and Control*, 53:165–190, 1982.

[Pnu83] Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the 15th Annual Symposium on the Theory of Computing*, pages 278–290, 1983.

[PZ84] Amir Pnueli and Lenore Zuck. Verification of multiprocess probabilistic protocols. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 12–27, 1984.

[PZ86] Amir Pnueli and Lenore Zuck. Verification of multiprocess protocols. *Distributed Computing*, 1:53–72, 1986.

[PZ89] Amir Pnueli and Lenore Zuck. Probabilistic verification by tableaux. Private Communication, 1989.

[Rab76] Michael O. Rabin. *Algorithms and Complexity*, chapter Probabilistic Algorithms, pages 21–40. Academic Press, New York, 1976.

[Rab82a] Michael O. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.

[Rab82b] Michael O. Rabin. N process synchronization with a 4 log_2 n-valued shared variable. *J. Comp. Syst. Sciences*, 25:66–75, 1982.

[Rao90] Josyula R. Rao. Eventual determinism: Using probabilistic means to achieve deterministic ends. Technical Report TR-90-08, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, 1990.

[Var85] Moshe Vardi. Automatic verification of concurrent probabilistic finite state programs. In *Proceedings of the 26th Symposium on the Foundations of Computer Science*, pages 327–338, 1985.

[Zuc86] Lenore Zuck. *Past Temporal Logic.* PhD thesis, The Weizmann Institute of Science, 1986.