UNIVERSITY OF GENEVA

ADVANCED FORMAL TOOLS

14X014

PRISM Case Study - Investor in Market

Authors:
Tansen RAHMAN*
Stephane NGUYEN*

E-mail:
Tansen.Rahman@etu.unige.ch
Stephane.Nguyen@etu.unige.ch

Author order determined by coin flip

March 2023

UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

Contents

1 Introduction **2**
1.1 PRISM . . . 2
1.2 PRISM Usage . . . 2
1.3 Futures Market Investor . . . 2
1.4 Motivations . . . 2

2 Background **3**
2.1 Models . . . 3
2.1.1 (Finite) Kripke Structure . . . 3
2.1.2 Discrete Time Markov Chain, absorbing states, extension with rewards . . . 4
2.1.3 Finite Markov Decision Process . . . 5
2.1.4 Turned-based stochastic game . . . 6
2.1.5 Our transition graphs . . . 6
2.2 Property Specification . . . 7
2.2.1 CTL: Computation Tree Logic . . . 7
2.2.2 CTL with state and path formulas . . . 8
2.2.3 PCTL: Probabilistic Computation Tree Logic . . . 9
2.2.4 Extensions . . . 10
2.3 Examples . . . 11

3 Implementation **14**
3.1 Model . . . 14

4 Property Verification **16**

5 Future Work **17**

6 Conclusion **17**

7 Appendix **18**
7.1 PRISM limitations . . . 18
7.2 PRISM Model Code . . . 19

1 Introduction

In this report, we implement a case study for the PRISM[5] model checking tool by extending an existing case study showcased on their website [1].

1.1 PRISM

PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour. It can be used to analyse systems from many different application domains, such as communication and multimedia protocols, randomised distributed algorithms, security protocols, and biological systems.

1.2 PRISM Usage

For learning how to use PRISM, we invite the reader to their well-documented tutorials and manual. A very useful tool from PRISM, called experiments, allows us, either from the GUI or CLI, to verify/check properties for different model constants/parameters in one go.

1.3 Futures Market Investor

We now discuss the original case study Futures Market Investor [1], found from their case studies list. The case study is based on an investor in a futures market. This example can be considered as a two-player game where one player (the investor) tries to maximize his return while the other player (the market) attempts to minimise the return of the investor. The below summarizes their description of the scenario:

"The Investor can make a single investment in 'futures': a fixed number of shares in a specific company that he can reserve on the first day of any month he chooses. Exactly one month later, the shares will be delivered and will collectively have a market value on that day – he or she can then sell the shares. The investor wants to make a reservation such that the sale has maximum value.

The market value v of the shares is a whole number of euros between €0 and €10 inclusive; it has a probability p of going up by €1 in any month, and 1-p of going down by €1, remaining within bounds. This probability represents short-term volatility.

The probability p itself varies month-by-month in steps of 0.1 between zero and one: p rises with probability 2/3 when v is less than €5, and when v is more than €5 the probability of p falling is 2/3. When v is €5 p rises or falls with equal probability. This represents market trends, where we assume the price of a stock revolves around its "true" value, here €5.

There is a cap c on the value of v, initially €10, which has probability 1/2 of falling by €1 in any month; otherwise it remains where it is. This models a company that is in a slow decline.

If in a given month the investor does not reserve, then at the very next month the market can temporarily bar the investor from reserving. But the market cannot bar the investor in two consecutive months."– *from the Futures Market Investor case study* [1]

Note that even though the case study could be considered a two-player game, their analysis focused on a Markov Decision Process with only one single player, the investor. The market's strategy is fixed in advance. Therefore, we can assume, for the rest of the discussion, that we will be dealing with a single agent/player.

1.4 Motivations

After observing the model and their description, we have some observations on their use of terminology and description of the scenario.

Firstly, this is not a representation of a Future stock, which is a contract to buy a stock at a pre-determined time and price. Instead, the investor is simply buying shares, that he receives with some delay. However, even this does not accurately match the model, as there is no cost related to buying these shares.

Secondly, after the investor receives the stock, there is an imposition that they sell the stock immediately, when more realistically they could wait further before selling them.

A proper scenario matching their model would be as follows. An investor has a number of shares. They would like to sell the shares, but there is a time delay before the sell is executed. They want to time the sell in order to maximize the return.

The original scenario has some room for possible extensions. In our implementation, we explore a few differences.

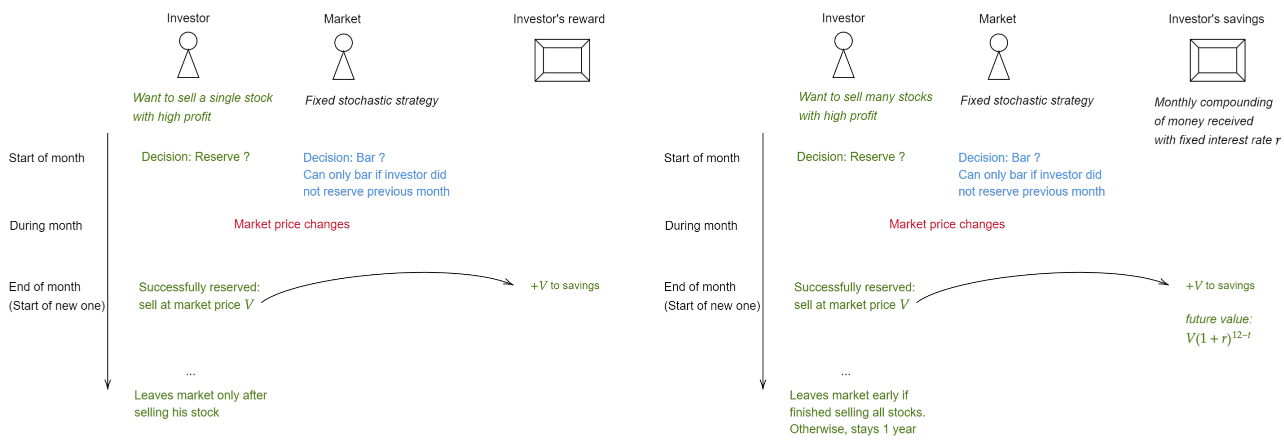
In the original model, the end condition is when the investor performs the single sell of his shares but there's a possibility of never reaching it, as the investor can choose to never reserve or sell. We change the end condition to a time-based one, permitting both the investor to continuously act in the market and the investor to leave the market after some time horizon. This of course would not make sense without any further changes, as the scenario has become such that the investor has access to an infinite amount of shares at his disposal. There needs to thus be a cost/limitation of some sort.

One implementation could be to add a cost (or negative reward) for performing the reservation. At first this seems to just become a lower bound for the stock value at which the investor would be willing to reserve his sell. However, if the cost is still considered even when the market bars the investor's sell that month, this variable would be interesting to observe, as it now is part of the dynamic between the two parties. However, PRISM at the moment strictly does not support costs (negative rewards). This idea is thus reserved for future work, since although for Markov Chains negative costs might be easier to implement, for Markov Decision Processes it is not as easy to handle.

Another way of implementing the limitation, one which we settled on, is simply to have a limit of shares that the investor has at hand.

Our last extension is to add a time-based reward to the sell of the shares. This is done by having the sell value accrue interest, compounded monthly. In other words, it's as if the received money is stored and grows with a fixed interest rate. This is in-line with the standard financial philosophy of "money now is worth more than money later".

The original model had 6688 states, and since we added two extra variables given at run-time, we have $6688 \cdot \text{tmax} \cdot \text{max\_stocks}$. **TODO: check this**



(a) Original case study

(b) Our case study

Figure 1: Diagram intuitively explaining the original case study versus ours

2 Background

We detail below some background knowledge that can be helpful. The reader can skip this section if the reader wants to directly dive into our model's case study. However, it's at least recommended to read the section 2.1.5 on our transition graphs and the section 2.3 on examples.

2.1 Models

A model is a representation, and usually a simplification of a real system, such that the focus is on the critical aspects. An example of a model is a transition system, which is a combination of states and transitions.

A state can represent, for example, variables of a program taking particular values. This state can change over time.

The transition from a state to another can be randomly triggered or can also be caused by the actions of the users. The transition can also happen every time an instruction is executed in a program.

We might want to check or verify whether our system follows some requirements. This generally requires a property specification language and property verification algorithms.

2.1.1 (Finite) Kripke Structure

A finite Kripke Structure is a extension of a transition system defined by $\mathcal{K} = \langle \mathcal{S}, S^0, \rightarrow, AP, L \rangle$ where:

- $\mathscr{S}$ finite set of states
- $S^0 \subseteq \mathscr{S}$ non-empty set of initial states
- $\rightarrow \subseteq \mathscr{S} \times \mathscr{S}$ left-total[1] binary relation on $\mathscr{S}$ representing the transitions
- AP the set of atomic propositions
- $L : \mathscr{S} \rightarrow \mathscr{P}(AP)$ state labelling function that labels each state with a set of atomic propositions that hold on that state

This model is non-deterministic as we don't necessarily know in which state we'll end up next when faced with many successor states. However, given a strategy to choose the next state when faced with many successor states (e.g. user(s) deciding which state to pick), the system becomes deterministic.

Relation to property specification: From a given state of the Finite Kripke Structure, we can form a tree of all the possible trajectories from that state. Each step down the tree corresponds to the one transition step in the Kripke Structure.

A property one could want to verify is whether a set of atomic propositions $\subseteq AP$ stay true on all the states in the future from a given state. We could also verify whether there exists at least one path/run/trajectory/execution in which all the encountered states satisfy a set of atomic propositions $\subseteq AP$.

Specifying such properties can be done with Computation Tree Logic (CTL) formulas (syntax). Verifying a CTL formula (semantics) would give us a set of states satisfying the CTL formula.

2.1.2 Discrete Time Markov Chain, absorbing states, extension with rewards

A DTMC is a probabilistic model where next states are chosen with some probability. We can also have states (terminal/absorbing) where we are guaranteed not to move away from, our values are then set in stone.

A Discrete Time Markov Chain (DTMC) [2] is more formally a memory-less discrete-time finite state space stochastic process, meaning a discrete-time finite state stochastic process (sequence of random variables $S_1, S_2, \ldots$) with states satisfying the Markov property:

- $\mathscr{S}$ finite set of states
- $s_0 \in \mathscr{S}$ the initial state
- $P[S_{t+1} = s' | S_t = s]$ the transition probability such that the states satisfy the Markov property:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_t, \ldots, S_0]$$

If the DTMC is homogeneous, meaning that the probability does not depend on time, we can write the transition probabilities in a state transition matrix.

Absorbing states: We can also introduce the notion of absorbing states as "terminal" states of the system in which there's a probability of 1 to stay in the state and a probability of 0 to exit the state:

$$\sum_{s' \neq s_{\text{absorbing}}} P[S_{t+1} = s' | S_t = s_{\text{absorbing}}] = 0, \quad P[S_{t+1} = s_{\text{absorbing}} | S_t = s_{\text{absorbing}}] = 1$$

This is useful when we want to define a finite time horizon process as in our case study.

Relation to property specification: In addition to our definition, we can define a set of atomic propositions AP and a state labelling function $L : \mathscr{S} \rightarrow \mathscr{P}(AP)$ just like in Finite Kripke structures. We can similarly verify properties based on those atomic propositions.

[1] each state has at least one child

Finite Markov Reward Process To make things more interesting, we can add the notion of rewards to Markov Processes (in our case DTMC). A reward can be attached to a state or to a transition.

So our DTMC with rewards can be defined as:

- $\mathscr{S}$ finite set of states
- $s_0 \in \mathscr{S}$ the initial state
- $P[S_{t+1} = s', R_{t+1} = r' | S_t = s]$[2] the probability to end up in a given state s' from state s and receive a particular reward r'.
- $\gamma \in [0, 1)$ discount factor[3] weighting the importance of immediate rewards versus future rewards. We'll keep it at $\gamma = 1$ for the rest of the document meaning that we're only working with the expected sum of future rewards.

We can for example estimate how "good" it is to be in a given state (see value functions in reinforcement learning) but it's still impossible to take any action in this environment.

Rewards can also be used to measure the expected number of steps from the initial state to an absorbing state. Note that we will only give a reward of 0 once we reach the absorbing state in order to not create infinitely positive or negative cumulative reward.

2.1.3 Finite Markov Decision Process

In a DTMC with or without rewards, all transitions are determined through probability. If that were not the case, however, then some actor (or player) must choose one transition. These transitions are then called actions. A Finite Markov Decision Process is then a DTMC with rewards where some actors must take decisions.

A finite MDP can then be defined with the following:

- $\mathscr{S}$ finite set of states
- $\mathscr{A}$ finite set of actions
- $s_0 \in \mathscr{S}$ the initial state
- $P[S_{t+1} = s', R_{t+1} = r' | S_t = s, A_t = a]$[4] the probability to end up in s' and receive a particular reward r' after taking action a from state s.
- $\gamma \in [0, 1)$ discount factor[5] weighting the importance of immediate rewards versus future rewards. We'll keep it at $\gamma = 1$ for the rest of the document meaning that we're only working with the expected sum of future rewards.

Actions introduce non-determinism. To resolve non-determinism, we need to give a policy/strategy $\pi : \mathscr{S} \times \mathscr{A} \to [0, 1]$ saying what is the probability to pick an action in a given state. Given a policy, the model falls back to a Markov Reward Process which is deterministic. The goal of an agent is to find a policy/strategy that might maximize expected cumulative future rewards, rewards given by the environment.

One question we might ask is what is the maximum we can earn from the initial state ?

Relation to Futures Market Investor case study: The Futures Market Investor case study was defined as a Markov Decision Process with two actors, the investor and the market. However, the market was given a fixed strategy in the PRISM model, therefore, removing its non-determinism and merging the actions of the market agent into the environment dynamics. We still have a Markov Decision Process, with the investor being the only actor/agent/player.

Relation to property specification: As before, we can define a set of atomic propositions AP and a state labelling function $L : \mathscr{S} \to \mathscr{P}(AP)$, with which we verify some properties.

[2] Up to our best knowledge, PRISM only supports special cases with deterministic rewards and not stochastic rewards

[3] Usually helpful for obtaining a finite expected discounted sum of rewards for continuous tasks (i.e. when can have infinite trajectories)

[4] PRISM only supports, up to our best knowledge, some sub-case of $P[S_{t+1} = s', R_{t+1} = r' | S_t = s, A_t = a]$, for example with positive deterministic rewards

[5] Usually helpful for obtaining a finite expected discounted sum of rewards for continuous tasks (i.e. when can have infinite trajectories)

2.1.4 Turned-based stochastic game

Turned-based stochastic games are extensions of MDP with multiple agents. They are essentially MDP in which states are attached to a finite number of players. Only one player/agent can act from a given state.

Therefore, a TSG is an MDP with a partition of the state-space $\mathscr{S}$ describing which agent can act from which state.

- $\langle \mathscr{S}, \mathscr{A}, s_0, P[S_{t+1} = s', R_{t+1} = r' | S_t = s, A_t = a] \rangle$ an MDP
- N agents/players and $P = \{\mathscr{S}_i\}_{i=1}^{N}$ a partition of $\mathscr{S}$

The goal for each agent is to learn a good policy $\pi_i : \mathscr{S}_i \times \mathscr{A} \to [0,1]$ where $\mathscr{S}_i$ represents the set of states in which the agent i can act.

This type of model could be used for future works but it also requires a more general property specification language that deals with TSGs [6] as well as a new tool, the PRISM-games tool.

2.1.5 Our transition graphs

We will represent, later on, our PRISM model with many graphs instead of one graph, the one would expect from MDPs[6].

We choose to represent our PRISM model with our transition graphs for many reasons such as these:

- Number of states can be extremely huge for our case study even though the PRISM code isn't huge (see Appendix)
- Clarify details one might miss
- Better understand the PRISM model (code)
- Visualize the original model and our extension
- Avoid unintentionally creating or destroying properties

Our graphical representation of the model is of intermediate level of detail as we not only remove unnecessary details by aggregating states, but also keep details that are related to PRISM code (e.g local variable names).

This intermediate level of detail can help visually explain the PRISM model.

Our graphical representation is made of multiple transition graphs, each corresponding to one PRISM module. Each transition graph of each PRISM module can be described informally with:

- A set of nodes where each can represent many states of the MDP.
- An initial node representing a set of states of the MDP that includes[7] the initial state of the MDP. This initial node also describes what values the local variables of the PRISM module take initially.
- Each node contains values of some or all local variables[8] that are satisfied in all the states related to the node.
- Each node can have additional information telling how a local variable is updated each time we traverse the node.
- Dashed or non-dashed directed edges (transitions) can represent many transitions in the MDP.
- Different quantities can be attached to directed edges:
  - In red: transition name
  - In gray: brief description to clarify what the transition means
  - In green: conditions to satisfy in order to move along the transition. These conditions can involve local variables from other modules[9].
  - In black: information telling how a local variable is updated each time we traverse the transition
  - In blue: probability to transition

[6] See section 2.3 for an example with the classical graphical representation of an MDP

[7] A set of states instead of the single initial state due to the so-called synchronization of PRISM modules

[8] The values of the local variables that are not described can be deduced from the values written in the initial node as well as from other nodes or transitions.

[9] e.g for module `barred`, there's no local variable `i` and for module `investor`, there's no local variable `b`

When there's a blue and green description/quantity over a transition, it means that we first need the green quantity to be satisfied before moving according to the blue quantity.

If the green quantity is satisfied, then we can move according to the blue transition probability. Otherwise we don't move.

- Dashed directed edges instead of solid ones are mostly there for perceptual convenience even though it can remind us the fact that when we come back to the same node at a later time, it does not necessarily mean that we come back to the same state in the MDP.
- Green-colored directed edge represents a directed edge with a green quantity/condition.

Recall that the basis for our graphical representation is the MDP PRISM model, meaning that there are nodes from which an agent has to decide which action to take. Without a given policy, the model is non-deterministic.

2.2 Property Specification

Recall that in general, we have this following scheme:

- Define model
- Define properties we would like to verify
- Verify properties

This subsection mostly deals with the syntax and semantics/meanings of the CTL and Probabilistic Computation Tree Logic (PCTL) formulas. This subsection does not cover the algorithms behind property verification.
This subsection also assumes that the reader is familiar with CTL even though we briefly recall formal descriptions.

Specifying properties for Finite Kripke Structures can be done with CTL formulas. However, CTL does not support properties dealing with probabilities which arises in probabilistic models such as DTMC. Although PRISM has its own property specification language which includes many property specification languages, we'll only describe PCTL, which is an extension of CTL and is used for DTMCs, as it gives some basic intuition on how the properties can be specified in PRISM.
The reader is also invited to read the PRISM documentation on property specification and reward-based properties in order to specify properties for MDPs as some properties cannot be specified due to the MDP non-determinism (e.g we cannot use `R=?`, we need to use `Rmax=?` or `Rmin=?`).
TODO: reformulate

2.2.1 CTL: Computation Tree Logic

Minimal syntax for specifying properties: Let's denote CTL to be the set of CTL formulas.

- $a \in AP \Rightarrow a \in CTL$
- $\Phi_1, \Phi_2 \in CTL \Rightarrow \neg\Phi_1, \Phi_1 \vee \Phi_2, \mathbf{EX}\Phi_1, \mathbf{EG}\Phi_1, \Phi_1\mathbf{EU}\Phi_2 \in CTL$

Semantics of minimal syntax: Let's denote by $[[\Phi]], \Phi \in CTL$ the set of states satisfying Φ for a Kripke structure. Let's also denote by ρ a run/path within the transition system: $\rho : \mathbb{N} \to \mathscr{S}$.

- $[[a]] = \nu(a)$ if we have $\nu : AP \to \mathscr{P}(S)$ that gives the set of states satisfying the atomic proposition a. Otherwise, one can use the labeling function L to retrieve it.
- $[[\neg\Phi_1]] = S \setminus [[\Phi_1]]$
- $[[\Phi_1 \vee \Phi_2]] = [[\Phi_1]] \cup [[\Phi_2]]$
- $[[\mathbf{EX}\Phi]] = pre_\exists([[\Phi]]) = \{s \in \mathscr{S} | \exists t \in \mathscr{S} \text{ s.t } s \to t \text{ and } t \in [[\Phi]]\}$ (predecessors of states satisfying Φ)
- $[[\mathbf{EG}\Phi]] = \nu Y.[[\Phi]] \cap pre_\exists(Y) = \{s \in \mathscr{S} | \exists \text{ run/path } \rho \text{ with } \rho(0) = s \text{ and } \rho(i) \in [[\Phi]], \forall i \geq 0\}$ (greatest fixed point)
- $[[\Phi_1\mathbf{EU}\Phi_2]] = \mu Y.[[\Phi_2]] \cup ([[\Phi_1]] \cap pre_\exists(Y)) = \{s \in \mathscr{S} | \exists \text{ run/path } \rho \text{ with } \rho(0) = s \text{ and } k \geq 0 \text{ s.t } \rho(i) \in [[\Phi_1]], \forall i < k \text{ and } \rho(k) \in [[\Phi_2]]\}$

Extended syntax: Let's define $\Phi_1 \equiv \Phi_2 \iff \forall \mathcal{K}, [[\Phi_1]]_{\mathcal{K}} = [[\Phi_2]]_{\mathcal{K}}$ (see SMV course. Set of states satisfying Φ_1 are the same as the ones satisfying Φ_2 for any Kripke structure)

- $true \equiv a \vee \neg a$
- $false \equiv \neg true$
- $\Phi_1 \wedge \Phi_2 \equiv \neg(\neg\Phi_1 \vee \neg\Phi_2)$
- $\mathbf{AX}\Phi \equiv \neg\mathbf{EX}\neg\Phi$
- $\mathbf{AG}\Phi \equiv \neg\mathbf{EF}\neg\Phi$
- $\mathbf{EF}\Phi \equiv true\mathbf{EU}\Phi$
- $\mathbf{AF}\Phi \equiv \neg\mathbf{EG}\neg\Phi$
- $\Phi_1\mathbf{AU}\Phi_2 \equiv (\Phi_1\mathbf{AW}\Phi_2) \wedge \mathbf{AF}\Phi_2$ (all (paths/runs/executions) strong until. Φ_2 holds eventually.)
- $\Phi_1\mathbf{AW}\Phi_2 \equiv \neg(\neg\Phi_2\mathbf{EU}\neg(\Phi_1 \vee \Phi_2))$
- $\Phi_1\mathbf{EW}\Phi_2 \equiv (\Phi_1\mathbf{EU}\Phi_2) \vee (\mathbf{EG}\Phi_1)$ (exist (path/run) weak until)

Note that we can extend it even further (e.g $\implies$). The semantic is not extended by the extended syntax.

Remarks: To remember the formulas, $\mathbf{E}X, \mathbf{E}F, \mathbf{E}G$ use $pre_\exists$ (exist a path) while $\mathbf{A}X, \mathbf{A}F, \mathbf{A}G$ use $pre_\forall$ (see SMV course). Each time we see F (finally), we use an **union** (so least fixed point) and each time we see a G (globally), we use an **intersection** (so greatest fixed point) (because we want something globally true on the whole path).

For **AU** or **EU**, we have a least fixed point so we use μ instead of ν.

2.2.2 CTL with state and path formulas

Minimal syntax: Inspired by a PRISM lecture on probabilistic temporal logic, our CTL minimal syntax can also be split into a set of state formulas $\mathcal{F}_S$ and a set of path formulas $\mathcal{F}_P$ where $CTL = \mathcal{F}_S$. This decomposition is useful later when we want to compare CTL with PCTL.

- $a \in AP \Rightarrow a \in \mathcal{F}_S$
- $\Phi_1, \Phi_2 \in \mathcal{F}_S \Rightarrow \neg\Phi_1, \Phi_1 \vee \Phi_2 \in \mathcal{F}_S$
- $\Psi \in \mathcal{F}_P \Rightarrow \mathbf{E}\Psi \in \mathcal{F}_S$
- $\Phi_1, \Phi_2 \in \mathcal{F}_S \Rightarrow \mathbf{X}\Phi_1, \mathbf{G}\Phi_1, \Phi_1\mathbf{U}\Phi_2 \in \mathcal{F}_P$

State formulas are for properties of states while path formulas are for properties of paths.

Semantics of minimal syntax: Splitting the minimal syntax leads to a rewriting of the semantics (but the meaning/semantics of the CTL formulas do not change). Let's recall that $[[\cdot]]$ represents a set of states. We also denote by $[[\Psi]]^P$ the set of paths that satisfy Ψ.

Inspired by the PRISM lecture on probabilistic temporal logic, our semantics can also be rewritten as:

- Semantics of state formulas:
  - $[[a]] = \nu(a)$ if we have $\nu : AP \to \mathcal{P}(S)$ that gives the set of states satisfying the atomic proposition a. Otherwise, one can use the labeling function L to retrieve it.
  - $[[\neg\Phi_1]] = S \setminus [[\Phi_1]]$
  - $[[\Phi_1 \vee \Phi_2]] = [[\Phi_1]] \cup [[\Phi_2]]$
  - $[[\mathbf{E}\Psi]] = \{s \in \mathcal{S} | \exists$ run/path ρ with $\rho(0) = s$ and $\rho \in [[\Psi]]^P\}, \quad \mathbf{E}\Psi \in \mathcal{F}_S$
- Semantics of path formulas:
  - $[[\mathbf{X}\Phi_1]]^P = \{\rho | \rho(1) \in [[\Phi_1]]\}, \quad \mathbf{X}\Phi_1 \in \mathcal{F}_P$
  - $[[\mathbf{G}\Phi_1]]^P = \{\rho | \rho(i) \in [[\Phi_1]], \forall i \geq 0\}, \quad \mathbf{G}\Phi_1 \in \mathcal{F}_P$
  - $[[\Phi_1\mathbf{U}\Phi_2]]^P = \{\rho | \exists k \geq 0$ s.t $\rho(i) \in [[\Phi_1]], \forall i < k$ and $\rho(k) \in [[\Phi_2]]\}, \quad \Phi_1\mathbf{U}\Phi_2 \in \mathcal{F}_P$

where $\Phi_1, \Phi_2 \in \mathcal{F}_S, a \in AP, \Psi \in \mathcal{F}_P$

2.2.3 PCTL: Probabilistic Computation Tree Logic

Specifying properties of DTMC can be done via PCTL. CTL cannot verify properties that has :

- A time component (something that holds within some period of time)
- Probabilities (working on probabilistic models)

So PCTL is an extension of CTL but with a probability that something is true within some time units. PCTL also applies to DTMCs instead of Finite Kripke Structure. PRISM's logics includes extensions of PCTL with rewards and with quantitative properties, properties that return numerical values.

TODO: verify that it's correct and finish this subsubsection on PCTL

Specifying properties using PCTL formulas: Our formulas are adapted or deduced from the paper of the authors of PCTL [4] in order to look similar to what is used in PRISM.

Minimal syntax: Let's denote $PCTL = \mathscr{F}_S \cup \mathscr{F}_P$ to be the set of PCTL formulas where $\mathscr{F}_S$ is the set of state formulas and $\mathscr{F}_P$ is the set of path formulas (see original paper [4])

- $a \in AP \Rightarrow a \in \mathscr{F}_S$
- $\Phi_1, \Phi_2 \in \mathscr{F}_S \Rightarrow \neg\Phi_1, \Phi_1 \vee \Phi_2 \in \mathscr{F}_S$
- $\Phi_1, \Phi_2 \in \mathscr{F}_S$ and $t \in \mathbb{N} \cup \{\infty\} \Rightarrow (\Phi_1 \mathbf{U}^{\leq t} \Phi_2), (\Phi_1 \mathbf{W}^{\leq t} \Phi_2) \in \mathscr{F}_P$[10]where **U** is for strong until and **W** for weak until[11]
- $\Psi \in \mathscr{F}_P$ and $p \in [0,1] \Rightarrow P_{\geq p}[\Psi], P_{>p}[\Psi] \in \mathscr{F}_S$

State formulas are for properties of states while path formulas are for properties of paths. Moreover, recall that we're working with Discrete Time Markov Chains so t represents t discrete time units.

Semantics of minimal syntax: Let's denote by $[[\Phi]], \Phi \in \mathscr{F}_S$ the set of states satisfying Φ and denote by $[[\Psi]]^P, \Psi \in \mathscr{F}_P$ the set of paths satisfying Ψ.

Our semantics are as follows and where in gray, we have parts that are similar to what we had in CTL with state and path formulas:

- Semantics of state formulas:
  - $[[a]] = v(a)$ if we have $v : AP \to \mathscr{P}(S)$ that gives the set of states satisfying the atomic proposition a. Otherwise, one can use the labeling function L to retrieve it.
  - $[[\neg\Phi_1]] = S \setminus [[\Phi_1]]$
  - $[[\Phi_1 \vee \Phi_2]] = [[\Phi_1]] \cup [[\Phi_2]]$
  - $[[P_{\geq p}[\Psi]]] = \{s \in \mathscr{S} | \Pr\left(\{\text{ run/path } \rho | \rho(0) = s \text{ and } \rho \in [[\Psi]]^P\}\right) \geq p\}, \quad P_{\geq p}[\Psi] \in \mathscr{F}_S$
  - $[[P_{>p}[\Psi]]] = \{s \in \mathscr{S} | \Pr\left(\{\text{ run/path } \rho | \rho(0) = s \text{ and } \rho \in [[\Psi]]^P\}\right) > p\}, \quad P_{>p}[\Psi] \in \mathscr{F}_S$ TODO: fix the notation
- Semantics of path formulas:
  - $[[\Phi_1 \mathbf{U}^{\leq t} \Phi_2]]^P = \{\rho | \exists k \geq 0 \text{ with } \boldsymbol{k \leq t} \text{ s.t } \rho(i) \in [[\Phi_1]], \forall i < k \text{ and } \rho(k) \in [[\Phi_2]]\}, \quad \Phi_1 \mathbf{U}^{\leq t} \Phi_2 \in \mathscr{F}_P$
  - $[[\Phi_1 \mathbf{W}^{\leq t} \Phi_2]]^P = [[\Phi_1 \mathbf{U}^{\leq t} \Phi_2]]^P \cup \{\rho | \rho(i) \in [[\Phi_1]], \forall i \geq 0 \text{ s.t. } \boldsymbol{i < t}\}, \quad \Phi_1 \mathbf{W}^{\leq t} \Phi_2 \in \mathscr{F}_P$

where $\Phi_1, \Phi_2 \in \mathscr{F}_S, a \in AP, \Psi \in \mathscr{F}_P$. Note that negation on probabilities "flips" the bound: e.g. $[[\neg P_{\geq p}[\Psi]]] = [[P_{<p}[\Psi]]]$

TODO: verify

Very brief example: A state s satisfying $P_{\geq p}[\Phi_1 \mathbf{U}^{\leq t} \Phi_2]$ means that there's at least a probability of p that Φ_2 will be satisfied within t time units (and that Φ_1 is satisfied in all the states in between).

[10] They are the so-called bounded variants of path properties. In PRISM, to get the unbounded version, we have to remove $\leq t$. Otherwise, in our formal description, we can have $t = \infty$

[11] The weak until does not require Φ_2 to hold

Relation with EX, EG and EU of CTL: Intuitively, from the brief example, we might already see a relation with CTL by varying that probability and time-bound t, what if $p > 0, t = \infty$? What if $p = 1, t = \infty$? A non-zero probability means that there exists a path while a probability of 1 means for all the paths. Let's denote by $\leftrightarrow$ the relation between PCTL and CTL even though they're not always completely similar. For instance, **TODO**

- $\mathbf{EX}\Phi \leftrightarrow P_{>0}[\Phi\mathbf{U}^{\leq 1} true] \vee \left(P_{>0}[\Phi\mathbf{U}^{\leq 1} true] \wedge P_{>0}[\neg\Phi\mathbf{U}^{\leq 0} true]\right)$ (Either current state satisfies Φ and a successor too or current state doesn't but a successor satisfies Φ)TODO: verify this
- $\mathbf{EG}\Phi \leftrightarrow P_{>0}[\Phi\mathbf{W}^{\leq\infty} false]$
- $\Phi_1\mathbf{EU}\Phi_2 \leftrightarrow P_{>0}[\Phi_1\mathbf{U}^{\leq\infty}\Phi_2]$

Relation with the extended syntax of CTL: We can write the following equivalences for the extended syntax. We omitted "true", "false" and "and" as they're defined similarly as for CTL.

- $\mathbf{AX}\Phi \leftrightarrow P_{\geq 1}[\Phi\mathbf{U}^{\leq 1} true] \vee \left(P_{\geq 1}[\Phi\mathbf{U}^{\leq 1} true] \wedge P_{\geq 1}[\neg\Phi\mathbf{U}^{\leq 0} true]\right)$ (Either current state satisfies Φ and all successors too or current state doesn't but all successors satisfy Φ)TODO: verify this
- $\mathbf{AG}\Phi \leftrightarrow P_{\geq 1}[\Phi\mathbf{W}^{\leq\infty} false]$
- $\mathbf{EF}\Phi \leftrightarrow P_{>0}[true\mathbf{U}^{\leq\infty}\Phi]$
- $\mathbf{AF}\Phi \leftrightarrow P_{\geq 1}[true\mathbf{U}^{\leq\infty}\Phi]$
- $\Phi_1\mathbf{AU}\Phi_2 \leftrightarrow P_{\geq 1}[\Phi_1\mathbf{U}^{\leq\infty}\Phi_2]$
- $\Phi_1\mathbf{AW}\Phi_2 \leftrightarrow P_{\geq 1}[\Phi_1\mathbf{W}^{\leq\infty}\Phi_2]$
- $\Phi_1\mathbf{EW}\Phi_2 \leftrightarrow P_{>0}[\Phi_1\mathbf{W}^{\leq\infty}\Phi_2]$

We could also, after defining the equivalences between the CTL minimal syntax and the PCTL syntax, just used the CTL equivalences between the CTL minimal syntax and its extended syntax.

The **E** (exists) and **A** (all) of CTL are hidden behind the probabilities. Essentially, to go from **E** (exist) to **A** (all), we just need to change the probability from > 0 to ≥ 1.

So what can we do more than CTL ?

- Can have a "portion" of the paths instead of at least 1 or all paths (More freedom on the quantification over paths).
- Can specify properties that hold during a specify time interval (More freedom on the quantification over time).

2.2.4 Extensions

Extension with rewards: Instead of verifying properties on probabilities, we can verify whether, for a DTMC extended with rewards, the expected sum of future rewards from the initial state, satisfies some bound. The DTMC extended with rewards can have many reward structures. Therefore, reward-based properties are conditioned on the reward structure. More information on different reward-based properties are described in their manual.

Extension with quantitative properties or rewards: We might sometimes want to ask what a particular probability or expected reward is instead of asking whether it satisfies some bound.

- $P_{=?}[\cdot]$, $P_{min=?}[\cdot]$ and $P_{max=?}[\cdot]$ for quantitative versions of P (See page 27 of [3] and the PRISM manual)
- $R^r_{=?}[\cdot]$, $R^r_{min=?}[\cdot]$ and $R^r_{max=?}[\cdot]$ for quantitative versions of R[12] (See page 30 of [3] and the PRISM manual)

These operators cannot be nested within PCTL formulas. They are the outermost operators.

The two quantitative properties $R^r_{min=?}[\cdot]$ and $R^r_{max=?}[\cdot]$ in which $\cdot$ is replaced by an expression representing the absorbing states, ask what are the worst and best expected cumulative reward obtainable (e.g in a MDP) from the initial state.

$$\min_{\pi} \mathbb{E}_{\pi,r}\left[\sum_{t=1}^{\infty} R_t \middle| S_0 = s_0\right], \quad \max_{\pi} \mathbb{E}_{\pi,r}\left[\sum_{t=1}^{\infty} R_t \middle| S_0 = s_0\right] \tag{1}$$

where π is the policy and r is the reward structure.

[12] The lower case r represents the reward structure.

Property specification in PRISM: PRISM property specification language syntax can differ from our definitions as it also uses the extended syntax with X, F, G etc. See their lecture on PCTL.

Here's an example of how we can specify that we use the reward structure named "steps":

```
R{"steps"}max=? [F is_done=1]
```

Some properties cannot be used for MDP due to its non-determinism. For instance, we cannot use `R=?` and `P=?`, we need to use `Rmax=?` or `Rmin=?` and `Pmax=?` or `Pmin=?` instead. Intuitively, we understood it as:

- We need to resolve the non-determinism by giving some strategy/policy as the behavior of the whole system depends on them.
- Min and max are related to two strategies, the worst one and the best one that would lead to the worst or best numerical value (dependant on what property we want to verify).

2.3 Examples

We now present examples from the formalism given above. We then show how they can be encoded in PRISM. This should give the reader an intuition on how DTMC's and MDP's in general are modeled in PRISM, and how to represent and verify PCTL properties in the PRISM tool.

Dice example: The dice example from PRISM tutorial Part 1, which tries to model a throw of a 6-face fair dice using fair coins, can be formally described via a DTMC:

- $\mathcal{S} \subseteq \{0,1,\ldots,7\} \times \{0,1,\ldots,6\}$ the set of reachable states where the first dimension represents `s` and the second dimension represents `d`, the dice number.
- $s_0 = (0,0) \in \mathcal{S}$ the initial state
- $P[S_{t+1} = s' | S_t = s] = T_{s,s'}$ the transition probability matrix (row stochastic matrix):

	(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)	(7, 1)	(7, 2)	(7, 3)	(7, 4)	(7, 5)	(7, 6)
(0, 0)	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0
(1, 0)	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0
(2, 0)	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0
(3, 0)	0	0.5	0	0	0	0	0	0.5	0	0	0	0	0
(4, 0)	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0
(5, 0)	0	0	0	0	0	0	0	0	0	0	0.5	0.5	0
(6, 0)	0	0	0.5	0	0	0	0	0	0	0	0	0	0.5
(7, 1)	0	0	0	0	0	0	0	1	0	0	0	0	0
(7, 2)	0	0	0	0	0	0	0	0	1	0	0	0	0
(7, 3)	0	0	0	0	0	0	0	0	0	1	0	0	0
(7, 4)	0	0	0	0	0	0	0	0	0	0	1	0	0
(7, 5)	0	0	0	0	0	0	0	0	0	0	0	1	0
(7, 6)	0	0	0	0	0	0	0	0	0	0	0	0	1

  where we didn't describe the unreachable states. $s = 7$ means that we have decided a dice number or in other words, that we have reached an absorbing state (or "terminal" state), hence the probability of 1 (in blue) to loop.

The DTMC graphical representation from the tutorial is shown in Figure 2a. The PRISM encoding or description of the model is:

```
dtmc

module die
    // local state
    s : [0..7] init 0;
    // value of the die
    d : [0..6] init 0;

    [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
    [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
    [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
    [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
    [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
    [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
    [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
    [] s=7 -> (s'=7);
endmodule
```

A property we can check is what is the probability to end up in an absorbing/"terminal" state with a particular dice number x by throwing fair coins successively and moving in our DTMC ? It should be around $\frac{1}{6}$ because it's a fair dice.

In PRISM, we can describe that we wish to compute quantitatively what that probability is by writing the following:

```
P=? [ F s=7 & d=x ]
```

where x is defined as a constant[13]. Running "Verify" would give us, using some iterative algorithm, a value for the probability. This PRISM formula is similar to what we would have written with the extension of PCTL with quantitative properties.

If instead of asking for the value, we wish to check whether the value of the probability satisfies some condition, for instance, if the probability is greater than 0.166, we can specify the PCTL formula $P_{>0.166}[true\,\mathbf{U}^{\leq\infty}(s=7)\wedge(d=x)]$ which corresponds to the following in PRISM:

```
P>0.166 [ F s=7 & d=x ]
```

and should return true.

A simple MDP example: Suppose we have an MDP where a player can choose to crouch or jump but has some probability to die depending on the actions chosen.

Let's say that the agent receives a reward of 1 for every time step without dying and a reward of 0 otherwise. Let's also say that crouching leads to a probability of living of 0.8 while jumping leads to a probability of living of 0.3. Clearly, the best action to pick in average would be to crouch.

- $\mathscr{S} = \{\text{dead}, \text{alive}\}$ finite set of states.
- $\mathscr{A}(\text{dead}) = \{\text{noop}\}, \mathscr{A}(\text{alive}) = \{\text{jump}, \text{crouch}\}$ finite set of actions
- $s_0 = \text{alive} \in \mathscr{S}$ the initial state
- $P[S_{t+1} = s', R_{t+1} = r' | S_t = s, A_t = a]$ the probability to end up in s' and receive a particular reward r' after taking action a from state s:
  - $P[S_{t+1} = \text{alive}, R_{t+1} = 1 | S_t = \text{alive}, A_t = \text{crouch}] = 0.8$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{alive}, A_t = \text{crouch}] = 0.2$
  - $P[S_{t+1} = \text{alive}, R_{t+1} = 1 | S_t = \text{alive}, A_t = \text{jump}] = 0.3$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{alive}, A_t = \text{jump}] = 0.7$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{dead}, A_t = \text{noop}] = 1$

  All the other probabilities are 0. Given a policy for our agent, it can stay alive for a longer period of time or not.

The MDP graphical representation is shown in Figure 2b. The PRISM similar encoding or similar description of the model is:

```
mdp

module mdp_example
    s: [0..1] init 1;  // alive initially

    // non-deterministic choice between jump and crouch from s=1
    [jump] (s=1) -> 0.3: (s'=1) + 0.7: (s'=0);
    [crouch] (s=1) -> 0.8: (s'=1) + 0.2: (s'=0);
    [noop] (s=0) -> (s'=0);  // absorbing states
endmodule

rewards
    (s=1) : 1;  // reward for staying alive
    (s=0) : 0;  // died
endrewards

rewards "steps"
    true : 1;
endrewards
```

However, it's not completely faithful to our formal definition of our model, as the rewards here are based on states instead of transitions. We need to remove 1 to the cumulative reward in order to get the correct expected sum of rewards.

Two quantitative properties we can ask, on this modified MDP, are what are the worst and best expected cumulative reward obtainable:

```
Rmin=? [ F s=0 ]
Rmax=? [ F s=0 ]
```

`Rmin=? [ F s=0 ]` gives us $1+\frac{0.3}{1-0.3}\approx 1.428571$ and `Rmax=? [ F s=0 ]` gives us[14] $1+\frac{0.8}{1-0.8}=5$.

Therefore, the worst and best expected cumulative reward obtainable in our original simple MDP example are $E_{min}=\frac{0.3}{1-0.3}\approx 0.428571$ and $E_{max}=\frac{0.8}{1-0.8}=4$ respectively.

These expected cumulative rewards correspond to the two cases:

[13] can be specified in the properties file using `const int x;`

[14] PRISM gave something very close to 5 but the exact value is 5

- When strategy is to only pick `jump` when alive so:

$$
\begin{aligned}
E_{min} &= 0.3 \cdot (E_{min} + 1) \\
(1 - 0.3)E_{min} &= 0.3 \\
E_{min} &= \frac{0.3}{1 - 0.3} \approx 0.428571
\end{aligned}
\tag{2}
$$

- When strategy is to only pick `crouch` when alive so:

$$
\begin{aligned}
E_{max} &= 0.8 \cdot (E_{max} + 1) \\
(1 - 0.8)E_{max} &= 0.8 \\
E_{max} &= \frac{0.8}{1 - 0.8} = 4
\end{aligned}
\tag{3}
$$

Our MDP example with a fixed strategy: If we fix the stochastic policy or strategy, meaning that $\pi(\text{alive}, \text{crouch}) = p_{\text{crouch}}$ and $\pi(\text{alive}, \text{jump}) = 1 - p_{\text{crouch}}$ are fixed, we fall back to a Markov Reward Process in which we can use PCTL to specify some properties.

The DTMC with rewards (Finite Markov Reward Process) is then defined by

- $\mathscr{S} = \{\text{dead}, \text{alive}\}$ finite set of states.
- $s_0 = \text{alive} \in \mathscr{S}$ the initial state
- $P[S_{t+1} = s', R_{t+1} = r' | S_t = s]$ the probability to end up in s' and receive a particular reward r' after moving from state s:
  - $P[S_{t+1} = \text{alive}, R_{t+1} = 1 | S_t = \text{alive}] = 0.8 \cdot p_{\text{crouch}}$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{alive}] = 0.2 \cdot p_{\text{crouch}}$
  - $P[S_{t+1} = \text{alive}, R_{t+1} = 1 | S_t = \text{alive}] = 0.3 \cdot (1 - p_{\text{crouch}})$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{alive}] = 0.7 \cdot (1 - p_{\text{crouch}})$
  - $P[S_{t+1} = \text{dead}, R_{t+1} = 0 | S_t = \text{dead}] = 1$

  All the other probabilities are 0.

The MRP graphical representation is left to the imagination of the reader. The PRISM similar encoding or similar description of the model is

```
dtmc

const double p_crouch;  // defines the policy

// MDP example with a given policy -> falls back to MRP
// p_crouch = 0.5 -> random policy
module mdp_example_random_policy
    s: [0..1] init 1;  // alive initially

    // Jump and crouch according to policy from s=1
    [jump_or_crouch] (s=1) -> 0.3*(1-p_crouch): (s'=1)  // jump
                + 0.7*(1-p_crouch): (s'=0)  // jump
                + 0.8*p_crouch: (s'=1)  // crouch
                + 0.2*p_crouch: (s'=0); // crouch
    [noop] (s=0) -> (s'=0);  // absorbing states
endmodule

rewards
    (s=1) : 1;  // reward for staying alive
    (s=0) : 0;  // died
endrewards

rewards "steps"
    true : 1;
endrewards
```

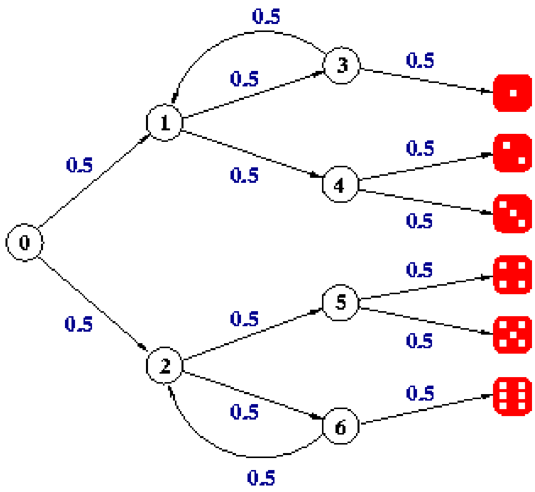
Again, it's called "similar" as the rewards for the default reward structure don't completely match, we need to subtract 1 from the cumulative reward in order to get the correct expected sum of rewards.

We show below three examples we can specify with CTL, PCTL, PRISM's logics respectively:

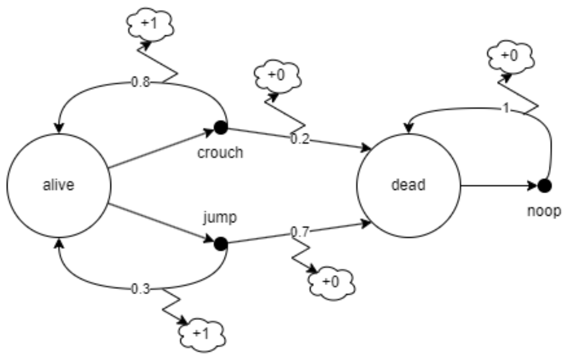
- CTL: Exist path where always alive: $\mathbf{EG}\, s = 1$
- PCTL (with extended syntax): Exist a path where alive for at least 10 steps: $P_{>0}[\mathbf{G}^{\leq 10}\, s = 1]$ (In PRISM, it's written as `P>0[G<=10 s=1]`)
- PRISM can get the value of that probability (quantitative property): `P=?[G<=10 s=1]`

and all of them, from the initial state $s_0 = \text{alive}$.

TODO: add example of `p_crouch` leading to some values



(a) Dice example, image taken from the tutorial



(b) Our simple MDP example

3 Implementation

3.1 Model

In the following, we represent our PRISM model using a graphical representation of the three main modules (month, investor, market) (See 2.1.5 for some explanation), textual descriptions of other modules (cap, value etc.) and textual description of the reward structures.

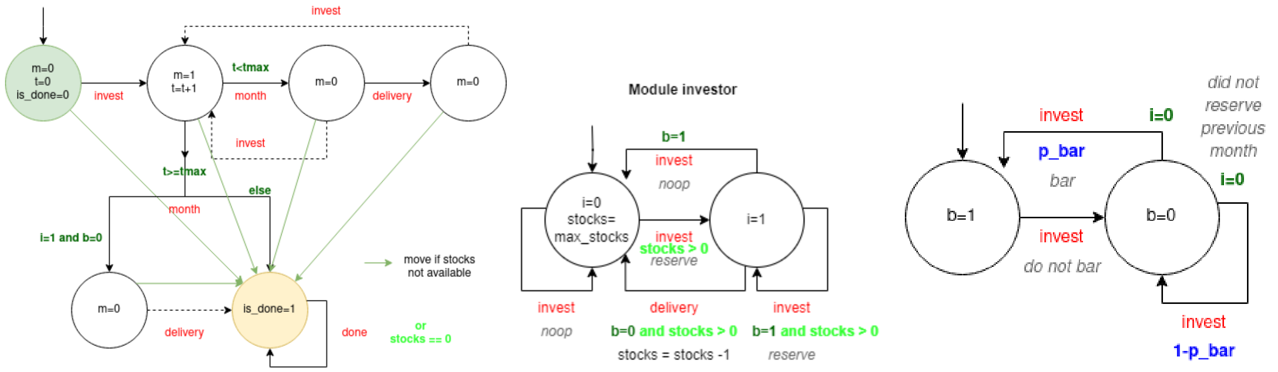
The model is parameterized by:

- `p_bar` : the probability of the market to bar
- `interest` : the interest rate from placing the money earned somewhere
- `v_init` : the initial market price
- `tmax` : the number of months
- `max_stocks` : the number of available stocks to sell

Changing these parameters can lead to very different quantitative properties as well as number of states of the MDP.

We can describe most of the model's behavior via our three main modules, explanation on how the other modules work and what are the different reward structures.

The month module, shown in Figure 3a, is the backbone module that the investor and barred/market modules will synchronize their actions with. It represents the passing of time with transitions occurring at the start, during, and end of each month. The variable `m` represents this, with `m=1` being the start of the month, `m=0` being the end, so the transition between the two is the "during". We have synchronized actions `[invest]` , `[delivery]` , and `[month]` for the aforementioned time of the month respectively. These will be discussed with their respective modules. The end condition is also monitored in this module, moving to the absorbing state with value `is_done=1` when the investor is out of stocks or when we reach the time limit.



(a) Month Module (b) Investor Module (c) Barred Module (Market module)

The investor module, as shown in Figure 3b but with an "abuse of notation"[15], represents the actions the investor takes, which are to reserve or not the sell at the beginning of the month, at synchronized action `[invest]`, and to perform the sell at the end of the month, at synchronized action `[delivery]`. Delivery also decrements the number of shares available to the investor, as they were just sold.

The barred/market module, as shown in Figure 3c, represents the actions the market takes, which is to either bar the investor from successfully reserving the sell or not. This is notably independent of the action the investor decides to take at the same time, since they are synchronized on `[invest]`, but the market can only bar if the investor did not invest the previous month. Note, however, that the model is still a Markov Decision Process, with one single player as the strategy of the market is fixed.

The remaining modules, for the value of the share, the probability of the value increasing/decreasing, and for the cap[16], update their values as described in subsection 1.3. The evolution of the value of this stock can be represented with a time series. An example is presented below.

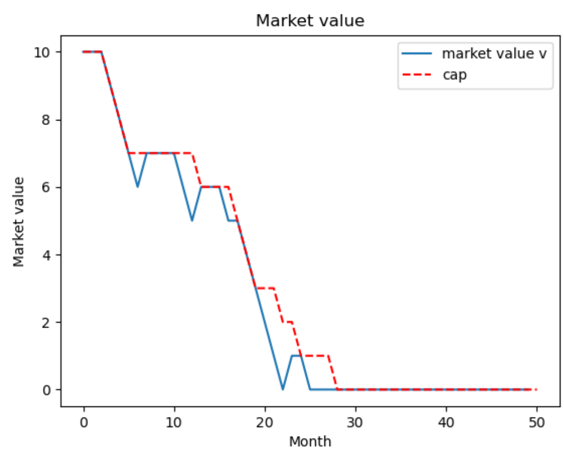


Figure 4: Example Time Series of the Stock Value. Note that the market value is not directly given as reward due to the `interest` parameter

Rewards: The model would be incomplete without reward structures. We define two reward structures:

1. Giving a reward of 1 for every transition except for transitions from absorbing states[17].
2. Giving a reward of $v \cdot (1 + \text{interest})^{\text{tmax}-\text{time}}$ every time a stock is sold/delivered (transition `[delivery]`) where v is the price of the stock for which the investor sold.

[15] It has an "abuse of notation" for the "initial node" as we don't necessarily always have `stocks=max_stocks` due to the `[delivery]` transition. Also recall that coming back to a node doesn't necessarily mean we go back to the same state in the MDP.

[16] ceiling of the value of the share that can decrease over time

[17] Transitions from absorbing states come back to absorbing states

The first reward structure is used to count the number of steps or transitions until reaching an absorbing state. The second reward structure is used to see how much the investor can earn, taking into account accrued interest with monthly compounding, given some policy.

4 Property Verification

In Figure 5a, we verify 4 quantitative properties within `tmax=12` months with no reinvestment into a bank or whatsoever (meaning `interest=0`) and `max_stocks=12` just to make sure we're never out of stocks to sell within the year (even though we need less):

- The first property (top left) shows the maximum cumulative reward/money the investor can get. We can see that the less likely the market bars, the higher the maximum reward was expected.
- The second property (top right) shows the minimum cumulative reward/money the investor can get. It's 0 because the investor can just do `noop` all the time. In other words, the investor never reserves. Note that it's independant of `v_init` and `p_bar` .
- The third property (bottom left) shows the maximum number of steps that the investor can get. We can see that it's independant of `v_init` but decreases as the `p_bar` increases. The maximum number of steps is always when the investor always reserves.
  In particular;
  - When the market never bars, the maximum number of steps is $3 \cdot \text{tmax} = 3 \cdot 12 = 36$.
  - When the market always bars when it can, the maximum number of steps is $2 \cdot \text{tmax}/2 + 3 \cdot \text{tmax}/2 = 12 + 18 = 30$ because the market bars only at second, fourth, sixth, ..., 12 month. Each time the market bars, it removes the delivery action/step.
- The fourth property (bottom right) shows the minimum number of steps that the investor can get. We can see that it's independant of both `v_init` and `p_bar` . The minimum number of steps is when the investor never reserves and the value is $2 \cdot \text{tmax} = 2 \cdot 12 = 24$. We can see some weird colors due to the slight deviation from 24 and this might be due to the iterative algorithm.

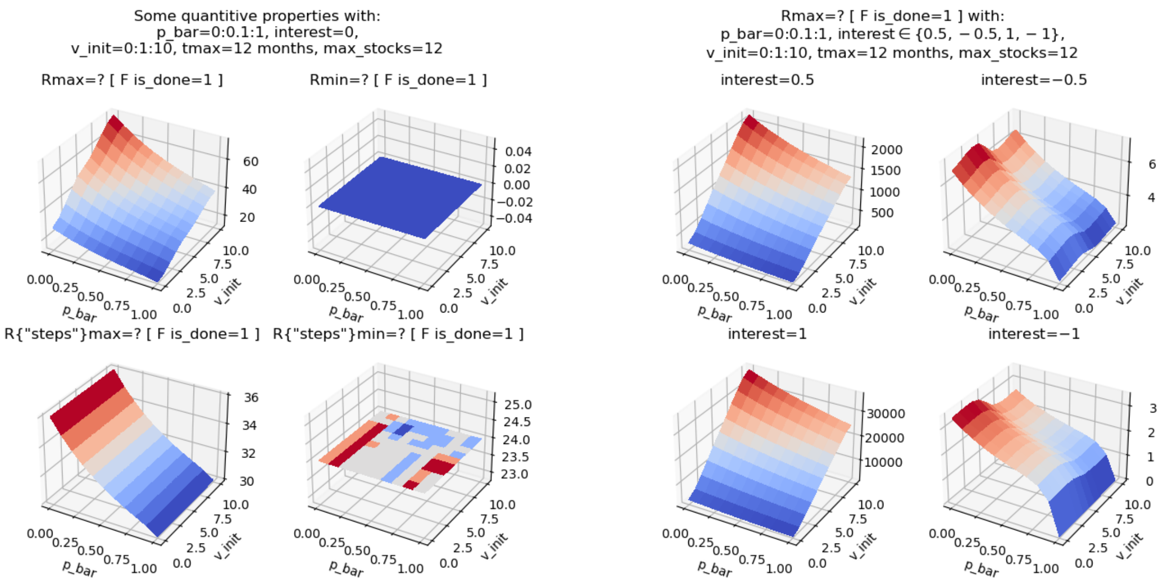
See our graphical representation of the module `month` in Figure 3a for more intuition.

In Figure 5b, we verify one property for different values of `interest` to observe its effect. All the other parameters are the same as previously. The property we want to verify is the maximum cumulative reward/money the investor can get, but this time, with a non-zero interest rate. As changing the interest rate doesn't affect the number of steps when the investor always stay for the `tmax` months due to our choice of `max_stocks` , we don't investigate the two properties based on the reward structure `"steps"` .

We can observe that a higher interest rate leads to a higher maximum expected cumulative reward. However, when the interest rate is negative, we see some harder-to-interpret wavy surface plots. This is caused by a combination of different factors:

- High `v_init` :
  - Cap can decrease over time and more likely that `v` initially **decreases** due to the market dynamics.
  - Trade-off between wanting to sell early due to the initial price fall & selling more stocks versus wanting to sell later due to losing money from the negative interest rate.
  - High negative impact of interest rate when we sell early.
- Low `v_init` :
  - Cap can decrease over time and more likely that `v` initially **increases** due to the market dynamics.
  - Trade-off between wanting to sell early due to selling more stocks versus wanting to sell later due to losing money from the negative interest rate & initial price growth.
  - Low negative impact of interest rate when we sell later.

In general, the incentive to sell early is no longer dominant and there's now a trade-off between selling early and selling later.



(a)

(b)

Figure 5: Verifying 4 quantitative properties for different parameters of the model. Parameters of the model include: `p_bar` : the probability of the market to bar, `interest` : the interest rate from placing the money earned somewhere, `v_init` : the initial market price, `tmax` : the number of months and `max_stocks` : the number of available stocks to sell

5 Future Work

Although we added the `max_stocks` variable, we leave the analysis as future work due to lack of time. Mainly, the effect of `max_stocks` on expected reward.

We could also look into the effect of both `interest` and `max_stocks` . When `max_stocks` is less than `t_max` , running out of stocks becomes an end condition. We can then look at when we reach this end condition. Notably, for a low initial value but a high interest rate, will all the stocks be sold early in the year due to the high interest rate, or will the investor wait for the value to rise before investing?

As noted in section 1.3, the original case study and our extension can be considered a two-player game, with both the investor and the market being independent actors. However, our project only focused on a single-player game, in which the market is given a fixed strategy.

The proper model to accurately represent the case study is a turned-based stochastic game, which was briefly touched upon in 2.1.4. Future work could look into implementing the original model and our extension in PRISM-games, which is an extension of PRISM with the ability to deal with TSGs. Further properties revolving around the dynamics between the investor and market could then be explored.

6 Conclusion

In conclusion, we delved into the formalism of state transitions, Markov Decision Processes, and Property Verification. We explored how these models and properties are encoded and executed in PRISM. Finally, we then take an existing case study, extend it and analyze properties. Future work could import these models as turn-based stochastic games into PRISM-games, and analyze further properties.

7 Appendix

7.1 PRISM limitations

Up to our best knowledge and understanding, we show below a non-exhaustive list of some PRISM limitations we encountered:

- No for loops, no lists or compact way to specify many similar transitions. See Bluetooth case study:

```
[reply]  receiver=2 & y1=0 -> 1/(maxr+1) : (receiver'=3) & (y1'=0) // reply and make random choice
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*1)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*2)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*3)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*4)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*5)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*6)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*7)
               [...]
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*122)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*123)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*124)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*125)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*126)
               + 1/(maxr+1) : (receiver'=3) & (y1'=2*127);
```

- No way to search/find elements (ctrl + f)
- No negative rewards possible for MDP
- No probability distribution over rewards.
- No direct way to specify/check a property on a transition based on a transition label: e.g where we only know the transition label from which we can reach a particular state but we don't know the properties that the state should satisfy. It's when we cannot identify the set of states.
- Variable ordering can affect performance due to the symbolic data structures and algorithms used in PRISM (Just like ordering matters in SFDDs).
- Inherent to the models, state-space explosion is an issue which can be further explored. PRISM includes symbolic data structures and algorithms (based on BDDs, MTBDDs) to help with state-space explosion just like SFDD can help for encoding Finite Kripke Structures in which the state labelling function is injective. Depending on the model, symbolic data structures doesn't necessarily help. In practice, when models become too big, one might want to work with function approximations (see deep reinforcement learning) even though it's uncertain about how one could verify properties.
- Limited graphing tools in PRISM: e.g we cannot have surface plots but we can just extract the results in CSV and use another programming language.
- Simulation in PRISM, when used for MDPs and when we specify for instance that we want the maximum cumulative reward or minimum cumulative reward, resolves the non-determinism counter-intuitively, using a uniform strategy. We would have expected the resolution to use either the best or worst strategy. However, using the worst or best strategy is not straightforward as one has to find them. In order to find them, there are iterative algorithms when the environment dynamics are fully known. But then it's no longer a complete simulation, it would become a simulation of a Markov Reward Process. Moreover, solving using the iterative algorithms should already give the state value function of the optimal policy, the maximum expected cumulative reward from each state and then following the optimal policy. Therefore, it means that we'll be doing extra work (the simulation) for nothing.

TODO: Fix references with websites and check for plagiarism

7.2 PRISM Model Code

Below we give the PRISM code for our extension of the case study. Recall that the original case study and its code can be found on their official website.

```
// EXTENDED "INVESTING IN THE FUTURES MARKET" from McIver and Morgan 03
// To finite horizon, multiple investments by Stephane NGUYEN and Tansen RAHMAN

mdp

const double p_bar;
const double interest;
const int v_init;
const int tmax;
const int max_stocks; // number of stocks investor has at the start

// module used to synchronize transitions (time)
module month

    m : [0..1];
    time : [0..tmax] init 0;
    is_done : [0..1] init 0;

    // Increment time, and perform transitions for the start of a new month (invest/bar).
    [invest] (m=0) & (time < tmax) -> (m'=1) & (time'=time+1);

    // Perform transitions that occur during the month (time series).
    [month] (m=1) & (time < tmax) & (is_done=0) -> (m'=0);

    // If last month, go to absorbing state, cashing in the shares if invested previous month
    [month] ((i=0)|((i=1) &(b=1))) & (m=1) & (time >= tmax) & (is_done=0) -> (is_done'=1);
    [month] (i=1) & (b=0) & (m=1) & (time >= tmax) & (is_done=0) -> (m'=0);

    // cash in shares
    [delivery] (m=0) & (time < tmax) & (is_done=0) -> (m'=0);
    [delivery] (m=0) & (time >= tmax) & (is_done=0) -> (is_done'=1);

    // two end conditions
    [done] (is_done=1) -> (is_done'=1);
    [done] (stocks=0) -> (is_done'=1);
endmodule

// the investor
module investor

    stocks: [0..max_stocks] init max_stocks; // number of stocks available to investor
    i : [0..1]; // i=0 no reservation and i=1 made reservation

    [invest] (i=0) -> (i'=0); // do nothing
    [invest] (i=0) & (stocks>0) -> (i'=1); // make reservation
    [invest] (i=1) & (b=1) -> (i'=0); // barred previous month: try again and do nothing
    [invest] (i=1) & (b=1) & (stocks>0) -> (i'=1); // barred previous month: make reservation
    [delivery] (i=1) & (b=0) & (stocks>0) -> (i'=0) & (stocks'=stocks-1); // cash in shares

endmodule

// market barring
module barred

    // b=0 - not barred and b=1 - barred, initially cannot bar
    b : [0..1] init 1;

    // do not bar this month
    [invest] (b=1) -> (b'=0);
    // bar this month (cannot have barred the previous month), only when investor did not invest last month
    [invest] (b=0) & (i=0) -> p_bar: (b'=1) + (1-p_bar): (b'=0);
    // case of b=0 and i=1 never happens because delivery would update i=0

endmodule

// value of the shares
module value

    v : [0..10] init v_init;

    [month] true -> p/10 : (v'=min(v+1,c)) + (1-p/10) : (v'=min(max(v-1,0),c));

endmodule

// probability of shares going up/down
module probability

    p : [0..10] init 5; // probabilitity is p/10 and initially the probability is 1/2

    [month] (v<5) -> 2/3 : (p'=min(p+1,10)) + 1/3 : (p'=max(p-1,0));
    [month] (v=5) -> 1/2 : (p'=min(p+1,10)) + 1/2 : (p'=max(p-1,0));
    [month] (v>5) -> 1/3 : (p'=min(p+1,10)) + 2/3 : (p'=max(p-1,0));

endmodule

// cap on the value of the shares
module cap

    c : [0..10] init 10; // cap on the shares
    // probability 1/2 the cap decreases
    [month] true -> 1/2 : (c'=max(c-1,0)) + 1/2 : (c'=c);

endmodule

rewards
    // reward from transition [delivery], accrued monthly interest
    [delivery] true : v * pow(1 + interest, tmax - time);
endrewards

rewards "steps"
    true : 1;
endrewards
```

References

[1] PRISM - Case Studies - Futures Market Investor.

[2] Markov chain, May 2023. Page Version ID: 1154313977.

[3] FOREJT, V., KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. Automated Verification Techniques for Probabilistic Systems. In *Formal Methods for Eternal Networked Software Systems*, M. Bernardo and V. Issarny, Eds., vol. 6659. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 53–113. Series Title: Lecture Notes in Computer Science.

[4] HANSSON, H., AND JONSSON, B. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing 6* (Feb. 1995).

[5] KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 585–591. Series Title: Lecture Notes in Computer Science.

[6] KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. Probabilistic Model Checking and Autonomy. *Annual Review of Control, Robotics, and Autonomous Systems 5*, 1 (2022), 385–410. _eprint: https://doi.org/10.1146/annurev-control-042820-010947.