PRISM Model Checker and Examples of Quantitative Verification

Kareem Khaled (400032153)

April 24, 2018

Abstract

This document highlights the importance of model checking and quantitative verification for software solutions. Its primary focus is to introduce the features of the PRISM modeling language and walk through the basics. A popular software failure in the healthcare industry is discussed, which will signify the need for verification of software. The structure of the document is as follows: This document first introduces the concept of PRISM and quantitative verification. It then walks through how to use the PRISM GUI and the tools it provides to assist in quantitative verification. Finally, the document applies verification techniques on a popular software failure in the healthcare industry: the Therac-25 Radiation Machine.

1 Link to Repository

This link contains a latex document along with the code referenced: PRISM Model Checker and Examples of Quantitative Verification

Contents

2 Introduction

In this section, PRISM and its features are discussed.

2.1 What is PRISM?

PRISM is a probabilistic model checker, meaning that it can model and analyse systems that contain probabilistic or randomized events. [4]

The PRISM model checker uses its own programming language called PRISM. It is a state-based language based on the Reactive Modules Formalism

2.2 What is Quantitative Verification?

Quantitative Verification is a technique for checking properties of a model relating to quantity, such as checking for a minimum number of cells needed in a cell cycle profile. [1]

2.3 What is Model Checking?

Model checking ensures that a model satisfies properties in a given specification.

There are 5 types of models that PRISM currently supports:

1. Discrete-time Markov Chains (DTMC), a deterministic model
2. Markov Decision Processes (MDP), a non-deterministic model
3. Continuous-time Markov Chains (CTMC), a rate based model
4. Probabilistic Automata (PA), another non-deterministic model
5. Probabilistic timed-automata (PTA), a real-time model

This document will cover probabilistic models called **Discrete-Time Markov Models**, or DTMCs. This model type calculates pure probabilities of an event occurring without non-determinism. DTMCs were selected for this document because they best exemplify the various features of PRISM, as it is the most supported model. A real-time non-deterministic model called **Probabilistic Timed Automata**, or PTA, will also be shown. PTAs are important because they can verify systems that depend on changes in time, which is the most relevant model for real-time systems.

More information on the different types of models can be found here: Model checking for probability and time: from theory to practice

2.4 Why PRISM is important

Between 1985 and 1987, the Therac-25 radiation machine (produced by Atomic Energy of Canada Limited) gave patients radiation overdoses of over 100 times the intended levels, which resulted in patient injuries and deaths. This is due to the producer's failure to test for software flaws in the system.

Had Atomic Energy of Canada Limited taken the time to map all possible states and behaviours with a model while ensuring safety properties are being met, this disaster in software development would not have occurred. [3]

PRISM is useful in proving that a piece of software achieves its intended properties, even if time is a factor that influences the result. PRISM's support for PTAs allows a tester to check the correctness of real-time software, or software with events that are constrained or influenced by time. This is especially important for companies that produce software that can affect a user's health.

2.5 PRISM in the industry

Some notable uses of PRISM are in the healthcare and network security industries:

- Cell Cycle Control in Eukaryotes
- Probabilistic Analysis of Network Anonymity
- Bluetooth Device Discovery

2.6 What PRISM cannot do

As of PRISM 4.4, PRISM's simulator (which allows for manual and automatic path explorations, and property simulations) does not support certain model types, including PTAs and models with multiple inital states.

Also, PRISM does not yet support the nesting of multiple temporal operators.

2.7 Installing PRISM

The latest version of PRISM as of April 24, 2018 is PRISM 4.4.

PRISM can be installed on Windows, Mac, Solaris, or Linux from the PRISM website: PRISM Model Checker

The only prerequisite software needed is Java version 8 or above.

3 PRISM GUI

In this section, the 4 main components of PRISM's GUI are introduced.
Once the PRISM GUI is open, there are 4 tabs visible on the bottom of the window. They are:

1. Model
2. Properties
3. Simulator
4. Log

3.1 Model

The **Model** tab is where a model to be represented using PRISM is created.

3.2 Properties

The **Properties** tab is where properties and probabilities for a model can be quantitatively verified and analysed. The output of probabilities in a simulation can be represented in PRISM with a graph.

3.3 Simulator

The **Simulator** tab is where steps in a model can be simulated. PRISM allows a user to manually or randomly walk through all possible states in our model.

3.4 Log

The **Log** tab is where all simulations and property checks of the current session in PRISM can be viewed. Any output, warnings, or errors will be visible in the log.

4 PRISM example: A 6 sided die

In this section, instructions for how to set up a PRISM model and the various uses of a model are presented. [4][5] A 6 sided die example is used because it is an intuitive example that can be represented as a DTMC, and thus introduces a user to many of PRISM's useful features.

4.1 Creating a model file

Ensure the current tab on the bottom of the PRISM window is 'Model'.

To create the die model, enter the following code into the empty text box:

```
dtmc

module die

//VARIABLE DECLARATIONS
        // current state (initialized to 0)
        s : [0..7] init 0;
        // current value of the die (initialized to 0)
        d : [0..6] init 0;

//COMMAND DECLARATIONS
        [] 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
```

The model can be saved using Ctrl S on Windows computers, or by hovering over Model in the toolbar at the top of the window and clicking 'Save Model'. Opening the file from the toolbar (Model -> Open Model) is also an option.

The file name of your model does not have to correspond with a module name in your code, as multiple modules can be declared in the same file. The file extension for DTMCs is .pm, but .prism also works for any model type. Save this file as a .pm file, as this example is a probabilistic model.

In this tutorial, the selected file name is die.pm

4.2 Model Syntax

4.2.1 Model Type Declaration

At the start of every model file, the type of model must be declared. In this example, a DTMC, or a Discrete-time Markov chains, is used. Hence, 'dtmc' is written at the top of the file.

4.2.2 Module Declaration

A model contains at least 1 module that may interact with outer modules in the file. The beginning of every module has a 'module (name)' declaration. To indicate the end of a module, 'endmodule' is used. Within a module, multiple variables and commands can be defined.

4.2.3 Variable Declaration

A variable must be declared within a module. Only 2 types are supported by PRISM: a range of integers and booleans.

A variable's name may consist of any letter, number, or an underscore.

In die.pm, 's', a range of integers from 0-7, is represented as:

```
s : [0..7] init 0;
```

This means that 's' can be within the range 0 and 7, and is initialized to 0.

Alternatively, 's' can be represented without an explicit initialization. PRISM assumes 's' begins at the lowest integer in the range, which is 0.

```
s : [0..7];
```

While die.pm does not contain a boolean variable declaration, a boolean can be represented with:

```
x : bool init false;
```

The initialization can also be omitted here, as it is false by default.

4.2.4 Command Declarations

```
[] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
```

The [] indicates the beginning of a command.

A guard is what follows the square brackets and is before the arrow. In the example above, the behaviour of the module when s=0 is described. Multiple guards in a single command are separated by an &.

(s'=1) and (s'=2) represents the two possible moves from state (s=0).

'0.5 : (s'=1)' represents the probability of s moving to 1 being 50%. The probability for each variable should add up to 100%, otherwise PRISM will notify the user of an error when the model is built.

The example states that s can move to either s=1 or s=2 with a 50% probability for each. The s' is necessary, as it indicates the next move for s.

If a user chooses to remain at the same state s=0, (s'=0) can be used to represent this. (s'=0) can also be omitted to keep the s at 0.

If a command simply moves to a new state after satisfying a single guard, a probability of 100% is assumed. An example of this scenario is as follows:

```
[] x=1 & y!=2 -> (x'=2);
```

This example is equivalent to:

```
[] x=1 & y!=2 -> 1:(x'=2);
```

Non-determinism was tested to work in a PRISM DTMC, but a warning is given in the log. This means that you are able to declare two commands with s=0 as a guard in a PRISM DTMC, but probabilistic models with non-determinism such as MDPs (Markov Decision Processes) and PTAs (Probabilistic Timed Automata) should be used instead. DTMCs are purely probabilistic, so adding non-determinism goes against the nature of the model and should not be done.

4.3 Simulator

Click on the 'Simulator' tab on the bottom of the GUI window. Right click anywhere on the window and select 'New Path'.

4.3.1 Manual Exploration

Manual Exploration allows for manual traversal across every reachable state. This is good for testing and debugging a model.

From the initial state, you can reach either state 1 or state 2 with the same probability (0.5, or 50%) . Under the Update header, click on s'=1 to move from the initial state to state 1.

From s'=1, select state 3 or state 4. Under the Update header, click s'=4 to move to state 4.

Now, click s'=7 and d'=3. Then click s'=7. This is our final state. By clicking s'=7 again, PRISM warns you that the selected path contains a **deterministic loop**.

This means that the previous state will be repeated by selecting this state. By disabling the deterministic loop detection, the only possible path is a continuous loop in the same final state.

This is pointless in our case, so click no to keep the deterministic loop detection enabled and to remain at the previous state.

The final value for s will be seven, as defined in die.pm

4.3.2 Automatic Exploration

Automatic exploration runs an automated simulation of a sample path with a defined number of steps.

After entering a number of steps, the next possible states are selected at random. A probability of which state will be selected next is shown under the 'Manual Exploration' box. Click 'Simulate' to move forward by your selected number of steps. It will stop once it reaches a deterministic loop or a Deadlock. Deadlocks will be discussed in the section below.

You can backtrack by a selected number of steps by clicking 'Backtrack'. This will undo the path taken and place you at the previous state. A new path will be selected if you click 'Simulate' again.

4.3.3 Deadlocks

Anytime the entire model is constructed, PRISM will notify the user if a deadlock is present. A deadlock occurs if the current state has no possible transitions. PRISM warns the user in the log that a deadlock exists and adds a self loop to that state. A deadlock either indicates that there is a problem in the model or in the system being represented by a model.

4.4 Properties

Properties allow us to quantitatively verify and analyse components of our model.

Select the tab on the bottom of the PRISM window labeled 'Properties'. The Properties tab allows the creation of properties to perform model checking and analysis on the created model.

There are 2 ways to add properties for our PRISM model. The first is through importing a .pctl file and the other is by adding properties directly into PRISM.

4.4.1 Creating a Properties File

In any text editor, create and save a file named die.pctl with the following text:

```
const int x; // a placeholder constant

P=? [ F s=7 & d=x ] // a Property definition
```

The file name does not have to correspond with the module name. The file extension can either be .pctl or .props

The extension '.pctl' can be used for DTMCs, MDPs, or PTAs while '.csl' can be used for CTMCs. '.props' can be used for any model type.

4.4.2 Importing a Properties File

In the toolbar at the top of the PRISM GUI, click Properties. In the dropdown menu, select 'Insert Properties List', then select the die.pctl file.

4.4.3 Creating a Property

Properties follow this general format:

```
P [ F phi ]
```

P is an operator that is given some bound.

phi is a path property.

F is a temporal operator that says: 'phi will occur at some point in this path'.

The likelihood of reaching a state s=7 (the final state) and some die value, d=x, in a path can be checked with:

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

The first component of this property is an operator 'P'. The 'P' represents a 'Probability'. In this case, 'P' is followed by '=?'. This means that the probability is transient. In other words, a probability is being searched for and an explicit bound (which would return a boolean value) is not defined.

This bound is asking for the probability of a specific path property occurring at some point in a path. The path property for this case is 's=7 & d=x', or when s is 7 and d is x. The value for x is defined by a user when they verify the property. x must be declared as a constant in the .pctl file created before being used, otherwise an error will be thrown by PRISM.

4.4.4 Verifying a Property

To verify a property, right click the property and select 'Verify'. A new window opens up stating the resulting probability. In the case where a placeholder constant is present, the user is asked to input a value for that placeholder.

To determine the percentage chance that the model will reach the die value 4, input '4' as the x value. The resulting probability is approximately 16.7 %.

4.4.5 Graphing Property Simulations

PRISM can generate a graph with resulting probability for a large number of path simulations.

Right click the screen on the properties page. Click 'New experiment'. Make sure 'Range', the bubble on the right hand side, is selected. This will allow for the selection of a range of x values and each of their corresponding values. Keep 'Start' at 0, set 'End' to be 7, and keep 'Step' at 1.

If the single value option (the bubble on the left hand side) were selected, there is no point in creating a graph, as there will be only 1 value in the graph (16.7%)

Before clicking 'Okay', check the box that says "Use Simulation". This tells PRISM to run the Experiment a set number of times.

The next screen is titled 'New Graph Series'. Ensure the Bubble 'New Graph' is filled on the next screen. A name for the graph can then be selected. Click 'Okay'.

The next screen allows for the setting of Simulation Parameters.

For now, use the default settings. Ensure 'Maximum path length' is set to 10000. Click Okay.

A Probability graph should now appear on the bottom right of the GUI under the selected graph name. The probability is not the same, because PRISM calculated the appearance of each die value for 10000 simulations. Random chance allows this probability to fluctuate.

Now, repeat the property graphing process, but increase the maximum path length to 100000. This will improve the accuracy of the probability results, because 100000 simulations as opposed to just 10000. The line looks similar to a plateau, as the probability of each die value is the same.

4.4.6 Additional Temporal Operators

PRISM is compatible with the following Temporal Operators:

- F : Is a path property true at some state in a path?
- X : Is a path property true at the next state in a path?
- U : Is a path property #1 true until exactly path property #2 is met?
- G : An invariant property, or one that stays true throughout all states in a path.
- W : The same as U, but path property #2 does not have to be met at some state in the model.
- R : The same as W inverted, or path property #2 is true forever.

More information on temporal logic can be found on: Temporal Logic
NOTE: PRISM does not currently support the nesting of Temporal operators.

4.4.7 Filters

PRISM allows a user to use **filters**, or to extract states that meet specific criteria while checking for a property. The resulting filter is presented in the log.

This is useful for finding the specific states that satisfy a property criteria.

Filters are created in the following format:

filter(op, property, states)

Where **op** is a filter operator,

property is any property,

and **states** is the filtering condition .

Filter operators that produce single values include:

- sum
- avg
- first
- forall
- exists

Other filter operators can produce multiple values:

- printall
- print

This document focuses on the **printall** filter.
More information on the other filters can be found here: PRISM Filters

The following is an example of the **printall** filter:

```
filter(printall, P=? [F true], s=7)
```

This filter shows all states that have an s value of 7 in the log. Notice that if a property condition is not used, a 'true' can be added in the square brackets.

In the log, all possible states which contain s=7 are presented alongside the resulting probability requested in the property. As (true) was the property condition, the following filtered results are outputted in the log:

```
Results (including zeros) for filter s=7:
7:(7,1)=1.0
8:(7,2)=1.0
9:(7,3)=1.0
10:(7,4)=1.0
11:(7,5)=1.0
12:(7,6)=1.0

Value in the initial state: 1.0

Time for model checking: 0.015 seconds.

Result: 1.0 (value in the initial state)
```

The probability of reaching any state is 100%, which is why the Result is 1.0. But notice how every item in the resulting filter contains an s=7. There are 1.0s beside each state filtered, as they each satisfy the condition (true)

If a property condition of (true) is replaced with (d=6) (the probability of a path covering die value 6), the following text is the result from the filter:

```
Results (including zeros) for filter s=7:
7:(7,1)=0.0
8:(7,2)=0.0
9:(7,3)=0.0
10:(7,4)=0.0
11:(7,5)=0.0
12:(7,6)=1.0

Value in the initial state: 0.16666650772094727

Time for model checking: 0.004 seconds.

Result: 0.16666650772094727 (value in the initial state)
```

Notice that each filtered state has a 0.0 with the exception of (7.6). This is because it satisfied the condition (d=6). The Result is now approximately 16.7%.

4.5 Rewards

The number of possible steps a path must go through before reaching a die value varies. In PRISM, this number is called 'reward', which essentially represents the cost of each step in a path.

4.5.1 Adding a Reward

Select the 'Model' tab at the bottom of the GUI, and add the following code after the 'end module' part of the code:

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

A reward can be named anything. 'steps' is defined to be the name of this reward. The name is not necessary, but adds clarity when more than one reward is defined.

For every step taken in a simulated path, 1 reward is shown beside 'steps'. The selection of the number of steps can be attributed to a condition. An example of this is:

```
rewards
    s=0 : 0;
    s=7 & d=2 : 2+s;
    d=6 : 100;
endrewards
```

Any defined reward must be bounded with 'endrewards'.

4.5.2 Summation of an Award

The summation of "steps" can be shown in the path by right clicking the "path" table, clicking 'Configure View', clicking Reward Visibility, then clicking
'1:"steps" (cumulative)'
Click the arrow pointing to the left, then click Okay. This makes the current summation of steps visible at every step in our path.

4.5.3 Reward Properties

Previously, the probability operator 'P' was discussed. 'R' is another property, representing 'Rewards'.

4.5.4 Adding a Reward Property

To add a new property, the existing die.pctl file can be directly modified then reloaded in PRISM. A faster alternative is to add the property directly in the PRISM GUI. In the 'Properties' tab, right click the 'Properties' box and select 'Add'. A popup window appears.

To ensure a program runs within 4 steps, this property can be added:

```
R{"steps"}>=3 [ F s=7]
```

Note that inside our bound, it is indicated to PRISM that the user want to check for a property with the specific reward "steps". This is useful when defining multiple rewards.

When verifying this property, true will be returned to the user if and only if the model's reward counter is at least 3 once any state containing 's=7' is reached. In this example, true is returned.

5 Quantitative Verification on the Therac-25 Radiation Machine

In this section, a high-level model of the Therac-25 Radiation Therapy Machine's software flaw is presented. [3][5]

5.1 Background

As discussed in the Introduction, PRISM can be used to model systems in the healthcare industry. A flaw in Therac-25 allowed the machine to administer radiation levels 100 times the intended level. This occurred specifically when the operator of the machine accidentally enabled 'Xray Mode', then quickly changed the Mode to the intended 'Electron Mode' for a patient. The high-current Xray electron beam would be administered to the client instead of the low-current electron beam.

It is apparent that the time the Therac-25 radiation machine took in switching modes and enabling a beam played a role in the system flaw. In order to model time alongside states, a PTA can be used.

5.2 Modeling Therac-25's flaw at a high level

Using PRISM, create a model that represents the described software flaw and name it Therac25.prism :

```
pta

module Therac25
p : [0..9] init 0; //Power level
Mode: [0..2] init 0; //Xray or electron mode
y : clock; //clock instance

//declare invariant

    invariant
        (Mode=1 => y>=5 ) &
        (Mode=2 => y>=3)
    endinvariant

[start] (Mode=0) -> 0.5: (Mode'=1)& (y'=3)&(p'=1) + 0.5: (Mode'=2)& (y'=5)&(p'=9) ;


//when in electron mode
[ElectronToXray] (Mode=1) ->  0.99 :(Mode' = 2) & (p'=9)& (y' = 3)
                            + 0.01 :(Mode' = 2) & (p'=1)& (y' = 1);
//when in Xray mode
[XrayToElectron] (Mode=2) ->  0.99 :(Mode' = 1) & (p'=1)& (y' = 5)
                            + 0.01 :(Mode' = 1) & (p'=9)& (y' = 1);

endmodule
```

5.2.1 Model Type

At the top of the .prism document, the 'pta' model type is declared. This allows for the creation of clock variables that can model real-time behaviour. This ability of modeling time is significant in quantitative verification, as it allows a user to model and verify real-time systems. An example of this is the Therac-25 Machine being modeled in this section.

5.2.2 Explanation of the Model

```
[XrayToElectron] (Mode=2) ->  0.99 :(Mode' = 1) & (p'=1)& (y' = 5)
                            + 0.01 :(Mode' = 1) & (p'=9)& (y' = 1);
```

The intended behaviour of Therac-25 is to enter Electron Mode with a power level of 1, a safe power level for patients. This occurs with a 99% probability and starts up at a time, 'y', of 5. There is a 1% chance for the machine to act erroneously when it is quickly changed to Electron

mode. In this case, the the transition is performed in time 'y' = 1. The power level is still at 9 from Xray mode, which is unsafe for patients.

5.2.3 Clocks

In Therac25.prism, a clock variable, 'y', is introduced. This timed automata allows a user to track and set time when transitioning to a state. Time is real-valued and constantly increasing.

PRISM does not currently support mathematical functions for clock values within a model. These functions, including mod; min; max; and log, can only be used for integers in PRISM. An error will appear when parsing the code if this is attempted.

5.2.4 Invariants

An **invariant** is a property or condition that must remain unchanged when some transformation is applied to a system.

In PRISM, invariants have coverage over 2 aspects of state[2]:

1. the automata can never transition into a state when an invariant is not satisfied
2. the automata can no longer remain in a state if the invariant is not satisfied

Invariant declarations in PRISM are only allowed in PTAs. An invariant is declared in the form:

```
invariant
    (Mode=1 => y>=3 ) &
    (Mode=2 => y>=5)
endinvariant
```

This invariant states that Mode=1 can only be entered while y>=3, and Mode=2 can only be entered while y>=5. The automata may remain in a specific Mode if the condition on time is satisfied. The invariant was selected for this model because it ensures the expected state of the radiation machine is reached when transitioning from Xray to Electron Modes. Due to the fact that an error in the machine has some relation to time, the invariant can help a user identify an error.

The invariant must be directly in between the variable declarations and commands of a module. Multiple invariants can be declared within this statement, separated by an '&'.

Clocks can be used within invariants to control the entry and exits of Modes in the model Therac25.prism . The presence of clocks in invariants are useful when checking properties that are influenced by time.

There is an intentional error in this system: the model does not always satisfy the invariant. This will be revealed in the next section.

5.3 Properties

5.3.1 Invalidating the Invariant

To begin, properties in PTAs do not support the 'P' and 'R' operators. Instead, Pmin, Pmax, Rmin, and Rmax should be used. They each return the minimum or maximum bound of either probabilities or rewards.

The goal is to declare a property that finds the maximum probability of a state with (Mode=2) and (p=9) occurring. In other words, this property checks to see if the machine can be in Electron Mode with a power of 9. This represents a hazardous flaw in our model, where a patient receives a radiation overdose. If this state is possible, the expected output is 1, or true.

Pmax=? [F(Mode = 1) & p = 9]

Try to verify this property. PRISM will alert you that it found a possible state that invalidates the invariant. The specific state(s) will be presented in the log.

By declaring an invariant and checking a property, a user discovers that there may be an error in their program.

A good question to ask is: Why did PRISM not notify ther user before that the model invalidated an invariant?

This is because previously, PRISM only parsed the model and caught syntactical errors.

Once a user requests PRISM to verify a certain property, it first generates the PTA diagram, then it constructs a reachability graph. In the reachability graph, PRISM checks for invalid states and notifies the user if the invariant isn't held in all states.

For testing purposes, the invariant can be commented out by adding a "//" before each line in it. Upon verifying the previous property, the probability of the unsafe state occurring is 1 (This means true, or the unsafe state is possible in some path).

This indicates to a programmer that there is either an error in the model created or the software being modeled is flawed.

5.3.2 Properties with Clock Variables

Before declaring a property that includes time, it is necessary to change the PTA model checking engine. This is because the default PRISM model checking engine, 'stochastic games', does not allow clock variables to be present in properties. If this is attempted, PRISM will state that there is an error.

To change the PTA model checking engine, click 'Options' on the toolbar located at the top of the GUI. Ensure the 'PRISM' tab is highlighted. Under the section 'PTA model checking method', change the method to 'Digital clocks'.

Now, clock variables can be used inside properties.

Two important notes for properties with clocks are:

1. Clock constrains cannot use the strict comparison operators > or <, but they can use >= or <= .
2. Two clock variables cannot be compared to each other.

With the invariant still commented out, add the property:

Pmax=? [F(y=1)]

This states: What is the maximum probability of a state occuring when the clock variable 'y' is equal to 1. Recall that (y=1) only occurs when the machine in unsafe state.

A boolean value 'true' is returned, as the unsafe state can be accessed with the model for the flawed Therac-25 Radiation Machine software.

5.4 Timelocks

When representing a PTA model in PRISM, it is possible to reach a state where no transitions can be made and time cannot elapse beyond a point specified in the invariant. This event is called a **timelock**. PTAs should not contain timelocks, so PRISM prompts the user about the error if it is encountered.

5.5 Simulations

As of PRISM 4.4, the simulator is not supported for PTA model types. This means that PRISM cannot generate a random path, manually walk through a path, or simulate properties, all of which were performed in the previous example on a 6-sided die.

6 Closing Words

In this section, the contents of this document will be summarized and further readings will be presented.

6.1 Final Remarks

The purpose of this document was to highlight the features of PRISM and discuss the benefits quantitative verification. Different models, including Probabilistic Timed-Automata and Discrete-time Markov Chains, were presented to the reader in examples. Properties of such models were then verified and analysed. Two topics relating to correctness (invariants and filters) were discussed alongside the model walkthroughs and quantitative verification examples. By the end of this document, a reader should see the importance of quantitative verification and model checking, as it can influence the safe development of any software and can potentially save lives.

6.2 Further Resources

The biggest inspiration for this document was a video by Professor Marta Kwiatkowska, from Oxford University. Although the video is lengthy, it is a great watch on the topic of quantitative verification.

The following is a good document that discusses temporal logic to a beginner. While PRISM doesn't support nested statements and every operator, it is still useful to be familiar with Temporal Logic

The following document discusses the various types of models in depth: Model checking for probability and time: from theory to practice

6.3 References

1. Kwiatkowska, M. (2007, September 3). Quantitative Verification. Retrieved from https://dl.acm.org/citation.cfm?id=1295018
2. Krause, C., & Giese, H. (n.d.). Model Checking Probabilistic Real-Time Properties for Service-Oriented Systems with Service Level Agreements. Retrieved from https://arxiv.org/pdf/1111.3110.pdf
3. Butler, D. (n.d.). Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator. Retrieved from http://users.csc.calpoly.edu/ jdalbey/SWE/Papers/THERAC25.html
4. Oxford University. (n.d.). PRISM Manual. Retrieved from http://www.prismmodelchecker.org/manual/
5. Oxford University. (n.d.). PRISM Tutorial. Retrieved from http://www.prismmodelchecker.org/tutorial/die