Bachelor thesis
Computing Science

Radboud University

A POMDP model for safety-critical systems and its deteriorating sensors

Author:
K. (Koen) Verdenius
s4361628

First supervisor/assessor:
Dr. N.H. (Nils) Jansen
N.Jansen@cs.ru.nl

Second assessor:
Prof. Dr. F.W. (Frits) Vaandrager
f.vaandrager@cs.ru.nl

July 2, 2021

Abstract

This thesis shows how to use Partially Observable Markov Decision Processes (POMDP) in industrially relevant and safety critical settings. A specific type of safety critical system will be described using a POMDP whose observations become less useful over time. This description is then turned into a model using the PRISM language. Furthermore, using several model checkers some properties of this model are examined.

Contents

Chapter 1

Introduction

One of the incredible things computing science has allowed us to do is solving real world problems. Models have been developed to solve Rubik's cubes, encryption problems and much more. Solving problems becomes a lot more difficult however once a combination of uncertainty, non-deterministic choices and probability becomes involved.

Problems such as Rubik's cubes are by no means easy to solve. What is however an alleviating factor when trying to find a solution is that outcomes are certain. When turning an edge to the left it is guaranteed what the new assortment of the Rubik's cube will look like. This certainty definitely makes for fun programming, however most things in the world do not have interactions with deterministic outcomes. Algorithms that interact with machines, people and animals have to deal with the inherent uncertainty that is brought on with this type of interaction. People do unexpected things, machines break down and even the best simulations do not always conform to the interactions an automated agent has when navigating outside of an controlled environment. Dealing with this uncertainty is very important however, since we do want systems that can fulfill a role in this world of ours. From simple things like allowing children to cross at a red light to complicated situations such as the control at a power station, dealing with uncertainty makes all of this possible.

A specific type of problems involving uncertainty are so called control problems [25]. These are the type of problems were the goal is optimization of a cost variable. This variable can be anything, such as energy spent, financial cost or an self defined cost function. As long as it is possible to quantify desirable outcomes. Possible control problems range from solving 2048 [27] to modeling financial systems [14]. Since optimization of cost functions is something we know how to deal with quite well, and a lot of problems can be formulated as a control problem, this provides a useful framework for

dealing with uncertainty.

Some of the problems one would want to model are so called safety-critical systems. Typically these are defined as systems were failure would involve significant loss of life, serious injury, property damage or environmental destruction [18]. When talking about these types of systems it is very important that we are sure some guarantees can be given about operations. One would want to know when to expect failure and its consequences. However, testing these systems in a contained environment is often incredibly expensive and sometimes not even conceivable. So how do we ensure that these systems are up to standards?

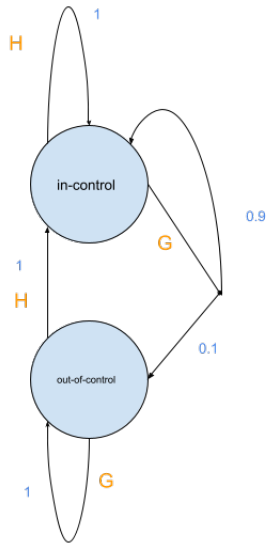
The solution is by simulating these type of systems in a formal model. This allows one to simulate the uncertainty the real world brings using probabilistic methods [19]. Although of course not a perfect representation of the real world, it gives us some insight in behavior, allows optimization of cost parameters and is a cheap and safe method for verifying otherwise prohibitively expensive and difficult properties a system might have.

There are many ways to model uncertain systems. One of the ways of dealing with this is the Markov Decision Process (MDP) [5]. This type of model allows a practitioner to develop a method for decision making were both non-deterministic choices are taken into account and were the unpredictability of their resulting outcomes are considered. An MDP consists of different possible states the system can take. Another part of this system are actions. Actions can be chosen to transition between states. These actions then have a set probability to transition the system to one or more other states. Transitions, states or both can provide a reward. Such a reward can alternatively be used as a cost. Once such a system is described, this information can be used to develop a policy. A policy is a set of actions taken from the current state. The outcome for actions is not certain, but one can take the uncertainty of actions into account when developing a policy. If you do this to optimize your expected reward, you are making an optimal policy. When developing such a policy, knowledge about previous actions or states of the system is unnecessary, since only information about current and future actions are needed to find an optimal set of actions from the current state of the system. This is the markovian porperty and guaranteed for all MDPs [31].

Lets try and flesh out this idea of an MDP with an example. The example is a similar but simpler version of the underlying MDP of the system we will describe in section 3.2 of this thesis. Imagine you have a safety-critical system. The system is either in-control of out-of-control. You can take an action `G` to continue operating. If you do this from in-control you

have probability of 0.9 to continue operating in-control and a probability of 0.1 to reach the out-of-control state. If you are in the out-of-control state action `G` will give you a probability of 1 to stay that way. Furthermore, there is also the option to action H. This will guarantee you to transition to the the in-control state no matter what state you are in. If we try to depict this as a visual model you wil get figure 1.1. Now lets say that if you end an

Figure 1.1: Example MDP



action in the control state or take action H, there is a cost of 10 incurred. Since we want to minimize the time spent in the out-of-control state, the policy we would design simply would be to minimize the cost. The way to do this is to minimize the expected cost the current action brings and to also minimize the expected cost future actions bring. This is a very basic description of Bellman's optimality principle which guarantees optimality for any MDP [6]. As this is a very basic system, a decent policy can be figured out intuitively. If you are in the in-control state, it is always a good idea for minimizing cost to take action `G` since action `H` guarantees a cost. Furthermore, you have a high probability of staying in the in-control state which again is good for minimizing cost in the near future. If you are in the out-of-control state, it is always a good idea to take action H. The cost is equal for taking action `H` or taking action `G` and staying in the out-of-control

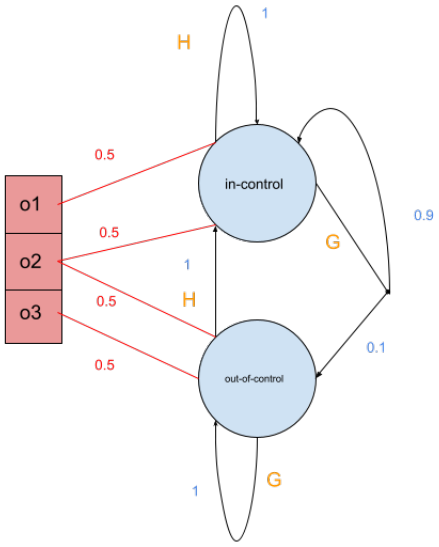
state, but you do have a lower expected cost in the in-control state in the near future. This concludes a very basic description of a policy. Do keep in mind that this is an almost stupidly simple system to introduce an MDP, and that for larger problems it is much more difficult to determine a good or even optimal line of actions to take.

Now lets move on to partial observability. Partially Observable Markov Decision Processes (POMDPs) [29] are an extended version of MDPs, The difference with MDPs is that knowledge about the current state cannot be directly observed and instead needs to be inferred through observations that provide partial information about what state the system currently resides in. Once such a model is described it again is possible to look for an optimal policy. This problem is know to be undecidable however. Therefore significantly more difficult to solve [21].

In order to understand this, the example MDP we presented will be extended. Nothing will change about the underlying MDP, so the model will still transition in the same way. The difference is that instead of seeing the states directly we now have three observables, o1, o2 and o3. If we are in the in-control state, there is a probability of 0.5 to observe o1 and a 0.5 probability to observe o2. If we are in the out-of-control state there is a probability of 0.5 to observe o2 and 0.5 to observe o3. The resulting POMDP is displayed in figure 1.2. To be clear, although the underlying MDP still transitions in exactly the same manner as before, we are unable to see the exact state of the system. All we have for information is the observations o1,o2 and o3. If we see observation o1 or o2, we have certainty that the system is either in-control of out-of-control. If we see o2, there is an equal chance that the system is in one of those states. If we now want to solve the problem of taking the appropriate actions, we have to base it solely on the observations available to us making the problem of finding an optimal or even an good solution significantly more difficult, especially for models more complicated then this simple example. One of the ways to solve POMDPs is to cast them to belief MDPs [17]. This is a specific type of continuous state MDP where you operate from a belief state, which is the likely state you are currently residing in. Value iteration can then be used with the current and future belief states to determine the best action to take.

As mentioned, optimality in POMDPs is undecidable. It is still worthwhile automating property verification though. This is done using model checkers. Verifying properties of a model automatically is done with the use of a model checker. A model checker is thus used to verify if a described system meets a given property. Examples of model checkers that are capable of model checking POMDPs are Prism [20] and Storm [13]. In order to perform model checking in these environments one would first need to

Figure 1.2: Example POMDP



describe their POMDP in a formal language designed to specify POMDPs. The PRISM language [26] is an example of such a language. Once such a model is described, the model checker can then be used to verify certain properties of the model or find optimal values for others.

Recently a system was described that has a safety-critical system and deteriorating sensors [30]. This comes down to a POMDP wherein the usefulness of the observations provided decreases over time. So when thinking back of the example described in figure 1.2, the probabilities of observing o1,o2 or o3 are no longer stagnant but change over time. This change will make it more difficult as operation continues to differentiate between the in-control state and the out-of-control state as time passes.

This thesis will cover all knowledge necessary for understanding the problem described in the previous paragraph. The original problem of a system with deteriorating sensors [30] will now be turned form a formal description into a much more descriptive depiction of what this POMDP actually looks like. This is then used to develop a generalized PRISM model for said problem and perform model checking on a few numerical examples.

Chapter 2

Preliminaries

2.1 Definition MDP

A Markov Decision Process (MDP) is a way of modeling non-deterministic choices that have a probabilistic transition. A slightly altered definition for MDPs can be derived from the book Principles of Modelchecking [5]. An MDP is a tuple M=$\langle S, A, P, R\rangle$ where:

- S is a finite set of states.
- A is a finite set of actions.
- P is a transition probability function defined as $P : S \times A \times S \to [0, 1]$ such that for all states $s \in S$ and actions $\alpha \in A$

$$\sum_{s' \in S} P(s, \alpha, s') = 1$$

- R is a reward function defined as $R : S \times A \to \mathbb{R}$

It should be mentioned for the purposes of this thesis that the reward function may also function as a cost function.

2.1.1 Solving MDP

It is possible at every state to take the optimal action to optimize the expected reward. A property of MDPs is that you do not need knowledge of previous states to do this for specific properties like expected reward or reachability. This is called the Markov property [31]. The way to do this is to get an optimal reward for the current state or action and an optimal expected reward for future states and actions. This idea is based in Bellman's principle of optimality and can be used to optimize MDPs. This leads to the following definition for an optimal policy [10].

- γ is an discount factor where $0 < \gamma < 1$
- If we want to minimize cost then

$$V(s) = \mathbf{min}_{\alpha \in A}(R(s, \alpha) + \gamma \sum_{s' \in S} P(s, \alpha, s')V(s'))$$

It is also possible to maximize reward. For that one simply needs to **max** instead of **min**. This definition is over an infinite horizon. Since models often loop and can be fairly large it is wise to limit the amount of iterations you use this value function for.

2.2 Definition POMDP

A Partially Observable Markov Decision Process (POMDP) is an extension of an MDP where, in contrast to a regular MDP, you do not have knowledge of the state and are only capable of deriving information about the current state using some observations. A definition for POMDPs can be derived from the chapter on POMDPs from the book Reinforcement Learning [29]. There, POMDPs are defined as a tuple M=$\langle S, A, \Omega, P, O, R \rangle$ where:

- S is a finite set of states.
- A is a finite set of actions.
- Ω is a finite set of observations.
- P is a transition probability function defined as $P : S \times A \times S \to [0, 1]$ such that for all states $s \in S$ and actions $\alpha \in A$

$$\sum_{s' \in S} P(s, \alpha, s') = 1$$

.

- O is an observation function defined as $O : S \times \Omega \to [0, 1]$ such that for all states $s \in S$

$$\sum_{y \in \Omega} O(s, y) = 1$$

.

- R is a reward function defined as $R : S \times A \to \mathbb{R}$.

It is still possible to try and develop a policy to maximize or minimize the expected reward. Unlike with an MDP this problem is undecidable [21].

2.3 Belief MDP

There are multiple strategies that can be used to attempt to solve POMDPs, and there are also many optimization strategies available for those strategies [8]. In this thesis no attempt will be made at solving a POMDP with a formal proof. It is still useful however to understand how a model checker tries to find a solution. Therefore it will be explained how a POMDP can be turned into a belief MDP [17] which then can be solved using value iteration. The specific solution methods of the model checkers used in this thesis are also mentioned in section 2.6.

A belief MDPs definition is similar to that of an MDPs definition. The difference is that instead of a set of states S the belief MDP has a set of belief states B. These belief states give a probability distribution over the states the system currently resides in, and are dependent on the POMDP you turn into a belief MDP. A belief state can be updated based on the action taken, the current observation and the previous belief state [32]. After taking action $\alpha \in A$ and reaching the next state s', observation $y \in \Omega$ is made. A belief state b is then a probability over state space S. The function $b(s)$ gives the probability that the the system currently resides in state s. The belief state is updated by:

$$b'(s') = \eta O(s', y) \sum_{s \in S} P(s, \alpha, s') b(s)$$

Where

$$\eta = 1/Pr(b, \alpha)$$

This Pr(b,a) is a normalizing constant defined as

$$Pr(b, \alpha) = \sum_{s' \in S} O(s', \alpha) \sum_{s \in S} P(s, \alpha, s') b(s)$$

Using these definitions a belief MDP [17] can be formally defined as a tuple M= $\langle B, A, \Upsilon, T \rangle$

- B is a set of belief states
- A is is a set of actions
- Υ is the reward function over belief states. This is defined as $\Upsilon : B \times A \to \mathbb{R}$ such that for all belief states $b \in B$ and actions $\alpha \in A$

$$\Upsilon(b, \alpha) = \sum_{s \in S} b(s) R(s, \alpha)$$

- T is a transition probability function defined as $T : S \times A \times S \rightarrow [0, 1]$

$$T(b, \alpha, b') = \sum_{o \in \Omega} \rho(b, \alpha, o) Pr(\alpha, b)$$

Remember that Ω is the set of observations. Furthermore:

$$\rho : B \times A \times O \rightarrow \{0, 1\}$$

Where

$$\rho(b, \alpha, o) \begin{cases} 1, & \text{if the belief update returns b'} \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

All of this can then be used to make an adapted version of the Bellman equation shown earlier, which then looks like this:

$$V(b) = \mathbf{min}_{\alpha \in A}(\Upsilon(b, \alpha) + \gamma \sum_{s' \in S} Pr(b, \alpha) V(s'))$$

Since there are uncountably many belief states this is generally unsolvable. There are, as mentioned, several ways of dealing with this. One of the more common is value iteration. This method partitions the belief space into a finite amount of regions by projecting the vectors of possible policies, which are sets of actions, onto the belief space [28]. These vectors are known to be piece-wise linear and conve. This, and the fact that these vectors converge over iterations can be used to determine the best policy given a certain amount of iterations. The best policy is determined by after projecting the vectors and partitioning the belief space into regions based on where each vector has the highest value, choosing the vector that yields the largest area over the belief space.

2.4 PRISM Language

The PRISM language is a simple language which can be used to develop a range of different state-based models. Among others, it can be used to develop both MDPs and POMDPs. In this section relevant knowledge for understanding the final model will be discussed. The examples discussed here can be found on or are based around information found on the publicly available manual for PRISM [2].

2.4.1 Models, modules and variables

Lets look at a first example:

```
mdp

module M1

    x : [0..2] init 0;

    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);

endmodule

module M2

    y : [0..2] init 0;

    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);

endmodule
```

At the top of the model, the type of the model is defined. PRISM allows for several options, although MDPs and further on POMDPs are the only ones worth mentioning for this thesis. After that there are two modules. Modules allow for two things: a definition of variables and commands. They begin with the "module" keyword and a name for the module, and end with "endmodule". Variables define the possible states the module can be in. Variables are made up of a name, an optional range and optional initial value.

2.4.2 Commands

For explaining commands, lets look at the following altered command from the example:

```
[a] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
```

Commands are made up out of several parts. The first is an optional action name, in this case a. These actions can be used if you want to assign transitional rewards, which will be explained later. Actions can also be used for synchronisation. Synchronisation happens if you have multiple modules and you have the same action for commands in different modules. During synchronisation these commands now occur simultaneously. This thesis does not use this synchronisation feature. After the action is the guard. This guard corresponds to the condition of the model at the start of a transition. Everything after the right arrow describes the possible states after transitioning and the corresponding probability of transitioning to that state. In this case for example if you are in state x=0 action [a] will have

an 80% chance of leading to state x=0 and a 20% chance of leading to state x=1. PRISM allows for local non-determinism meaning that if we write:

```
[] x=0 -> 1:(x'=0);
[] x=0 -> 1:(x'=1);
```

Both x=0 and x=1 are reachable from x=0.

2.4.3 Constants

Besides the definition of modules, PRISM also allows you to define global constants outside of modules:

```
const int radius = 12;
const double pi = 3.141592;
const double area = pi * radius * radius;
const bool yes = true;
```

As the name suggests, the value of these cannot be changed by commands. They can be used anywhere a value would be expected.

2.4.4 Rewards

The next piece of information important for understanding how MDPs, or specifically the MDP presented in this thesis, work in PRISM is understanding how rewards work. Lets look at the following example:

```
rewards
    x=1 : 1;
    [] true : some_constant;
    [b] x=0 : 2;
endrewards
```

Rewards can be awarded to states and to transitions. Rewards have an optional label, a condition and the corresponding reward. Rewards without a label are given if the model is in a state matching the condition. In the example above, x=1 is a state specific reward. For any state x=1, the reward is 1. Any reward with a label, empty or not, is rewarded upon a transition. An empty label gives its reward without a transition label. In this case the empty label has true as condition, giving all transitions without a label some_constant as its reward. Non-empty labels apply to transitions with a matching labels, its condition is applied to the state it starts its transition in. Transitions labeled b starting in x=0 will be rewarded 2 in this model.

2.4.5 Labels

Labels are of little consequence to the actual functioning of the model. They are used as a method for identifying a state or a set of states.

```
label "end" = time==maxtime;
label "late" = time>=10;
```

In the example above the label "end" applies to all states were the time variable equals the maximum time variable. The label "late" applies to all states were the time variable is greater then or equal to ten. Although labels are not of consequence to the workings of the model they are still important to mention since the environments for model checking POMDPs seem unable to identify state properties directly and instead need labels to identify said properties.

2.4.6 POMDPs and observables

Now that we understand how MDPs function, it is time to move on to POMDPs in PRISM. As explained previously the difference is that only some of the information about the state is observable. The way this is done in the PRISM language is to indicate which of the variables can be observed. All the other variables are still part transitions but simply cannot be used in making policy choices when solving the model. Lets adapt the MDP discussed previously into an POMDP. This looks as follows:

```
pomdp

observables
    x
endobservables

module M1

    x : [0..2] init 0;

    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);

endmodule

module M2

    y : [0..2] init 0;

    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);

endmodule
```

As you can see there are two main differences. The first one is that the model now begins with the keyword "pomdp" instead of "mdp", indicating that it is in fact a POMDP. The second is that one now needs to define which variables are actually observable when solving the model. In this case only x is observable, but as many variables as necessary can be chosen as long as they are separated with a comma. It is also possible to only have a

range of a variable be observable. This looks as follows:

```
observable "pos" = l>0;
```

For this thesis this option will not be used.

2.5 Properties in PRISM

CTL stands for computation tree logic, a temporal branching logic that can be used to verify properties in systems with non-deterministic choices [5]. PCTL stands for probabilistic computation tree logic and is an extension to CTL in that it allows verification of systems with non-deterministic choices that have a probabilistic transitions [12]. Property verification in the PRISM language is based on PCTL. Without going into to much detail in this section about what PCTL is, a quick understanding will be given about how property verification works in PRISM.

2.5.1 Queries and operators

Any property one wants to verify takes the following basic shape:

```
query [pathproperty]
```

or

```
operator bound [pathproperty]
```

This thesis makes use of three properties. Two of them follow the query structure, and one the operator bound structure.
The first propertythat follows the query structure used in this thesis looks as follows:

```
Rmin=? [pathproperty]
```

This traverses the model returns the minimum expected reward for a given path property. One can also maximize the property with Rmax. Typically this is *the* query of interest when verifying properties about a POMDP since minimization (or maximization) of a reward is often the main goal of a model.
The second property with a query structure looks as follows:

```
Pmin=? [pathproperty]
```

This traverses the model and returns the minimum expected probability for a given path property to be reached. One can also maximize the property with Pmax. This property is useful since not all model checkers are capable of checking a bound, which is the next property we will discuss. This problem can then be alleviated by simply using this property where you would otherwise use a bound and seeing if it matches what you were trying to check. The third property and the only operator bound one used in this thesis is the P operator. The P operator also called the probability operator. The operator takes the following shape:

```
P bound [pathproperty]
```

The operator holds true if the probability of the path-property occurring meets the condition set by the bound. So for example:

```
P>0.2 [pathproperty]
```

Means that the probability that pathproperty is satisfied by the path that starts from state s is greater than 0.2. This bound can be any atomic proposition.

2.5.2 Path property

There is only one path property of interest to us. The finalize or eventually property. An expression for the finalize property would look something like this:

```
F "Some state"
```

This property holds if the model eventually reaches a state labeled "Final state". So eventually is true in the initial state if for all possible paths, with all possible probabilities of transitioning, there is a path in which eventually the state is reached labeled some state.

2.6 Verification environments

The POMDP described in this thesis was developed in the PRISM language. There are different ways of actually model checking the POMDP. The three tools used in this thesis were Storm [13], Prism Model Checker [20] and a Toolchain that relies on Storm developed at the Radboud university, UC Berkeley and UT Austin[9]. On the websites of Storm [3] and Prism Model checker[1] detailed instructions are available if you plan on using these model checkers. For the tool-chain a link is included in the appendix to a Github repository which includes the tool chain and installation instructions.

2.6.1 Storm

Storm [13] is a model checker for models involving random or probabilistic phenomenon. Storm was benchmarked [11] to be the fastest model checker currently available. For this thesis the interest lies with its ability to model check POMDPs in the PRISM language. Furthermore it's python API allows one to develop their own program to interact with the different modules Storm has on offer. The Method Storm uses to solve POMDPs is an abstraction-refinement framework that uses discretised belief MDPs [7], resulting in bounds of the actual value of a policy. Model checking a POMDP in Storm is done by issuing the following command in your console:

```
./Storm-pomdp --prism [path to model] --prop [path to props
    file] --belief-exploration
```

The property file is simply a file containing the properties you want to model check. It is also possible to fill in a singular property you want to model check instead of a path to a property file.

2.6.2 Prism Model Checker

As of the latest update (4.7) Prism Model Checker [20] also has the ability to verify properties of POMDP's. The method Prism Model Checker uses by default for its verification is called value iteration and grid-based methods [24]. The way it works is by starting at the end point and refining an estimate of the given property by repeatedly working its way backwards. The user can either define a maximum number of iterations or Prism will choose one. This has a similar effect to the method used by Storm in that the final answer provided is an estimation and not an absolute solution. Model checking in Prism is done by issuing the following command in your console:

```
./prism [path to model] [path to props file]
```

2.6.3 Toolchain Radboud University

The final method used for verification is with the Toolchain Developed here at the Radboud University [9]. The Toolchain works by turning the POMDP into a parametric Markov chain [16] and then performing parameter synthesis to solve this chain [15]. For operators and their bounds the tool also guarantees correctness because it model checks the result verifying the solution. This means that when checking a threshold correctness is guaranteed. This guarantee is not possible when minimizing or maximizing a value however. Running the model is done by editing the solver.py file to the model and property you want to check and then running solver.py.

Chapter 3

Research

3.1 Example problem

In order to get a more concrete idea of what the model that will be described can be used for, an example will be sketched to give some insight.

An offshore wind turbine is a way of generating renewable energy that is becoming more and more common. As with most other forms of renewable energy, it does not require a constant input of resources to continue operating. It does require maintenance. The advantages of an offshore turbine compared to one on land is that, in general, offshore counterparts produce more power. The disadvantage is that maintenance is a lot more difficult and costly. It is also more costly to do a manual checkup. A way to mitigate this would be to install a sensor on the turbine to minimize visits. Such a sensor would give some indication about the disposition of the turbine, although it would not result in perfect knowledge. The Turbines disposition is usually in control, meaning it is operational and not costing anything because of this. Every so often it breaks down resulting in it being out of control.

When in ownership of an offshore wind turbine like this, it is incredibly important to make deliberate decisions about how to minimize or maximize certain attributes as to get the most out of your asset. An obvious goal could be to minimize operational costs, although less obvious goals could also be to minimize maintenance visits over a certain period or maximizing the total time the system spends in a state of being operational.

The outcome of these properties could be found by describing the system as a POMDP. This could be done by describing the disposition of the turbine (in-control or out-of-control) combined with the time as the underlying state, taking the imperfect sensor observation as the observation function and applying a cost to maintenance visits and the turbine being not oper-

ational. Actions between transitions would then be to continue operations or to perform a maintenance visit at a cost.

This example can be described using a "regular" POMDP. In fact, it has been done so before [22] [23]. Those solutions however do not take into account that not only the turbine deteriorates but that the sensor also deteriorates.

If the owners of the wind turbine would want to model the system taking this into account, they would have to model the system in such a way that over time observation provide less useful information until sensors are replaced. This adds the action of replacing the sensors to the possible options. If they would want to minimize a property such as operational costs, the solution would now have to balance costly maintenance of the turbine and very costly maintenance of the sensors with the risk of operating the system in an out of control state. As will be shown in this thesis, this can be achieved with a POMDP whose observation function is affected by the time spend operating the system.

3.2 Description of the model

Since this is not a thesis about wind turbines but about POMDPs with deteriorating sensors, this section will be used to abstract away from the example and give a description of the model. Some further explanation will also be included. This model is based on the paper [30]. Figure 3.1 also serves as a visual aide for understanding the model described.

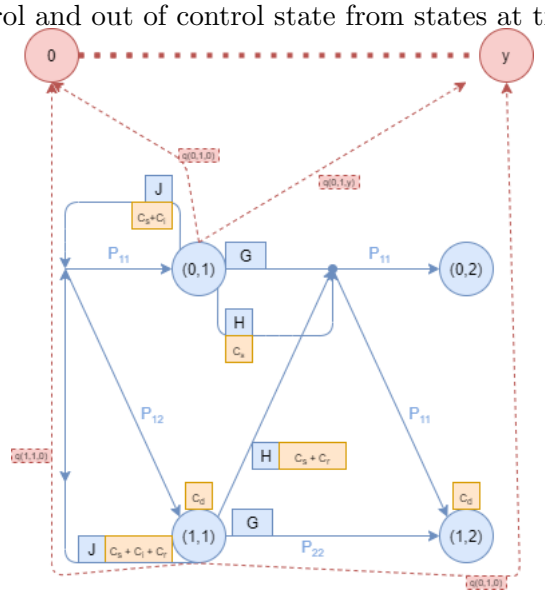
3.2.1 The state space

The following describes the state space of the POMDP:

- A disposition i ∈ D of the system is either in-control which corresponds with "0" or out of control which corresponds with "1"
- Periods are indexed by the set $\mathbb{N}$
- The underlying state of the system is made up of the pair $(i, t) \in D \times \mathbb{N}$

The disposition can be either in an in control "0" or an out of control state "1". A cost is incurred for being out-of-control. It is a discrete time system over the interval {0,1,.,n}. An upper limit n should be chosen to properly define the observation function later on. The states of the POMDP constitute a combination of the disposition, these states are displayed as the blue circles in figure 3.1.

Figure 3.1: A part of the model showing all the possible transitions from both an in control and out of control state from states at time interval 1



3.2.2 The actions

The following describes the actions of the POMDP:

- The action space is A={`G`,`H`,`J`}. Action `G` is continue operation, action `H` is full inspection without sensor replacement, Action `J` is full inspection with sensor replacement.
- If action `G` is taken a normal transition takes place as described by the transition function.
- If action `H` is taken, it is as if action `G` takes place from state (1,t).
- If action `J` is taken, it is as if action `G` takes place from state (1,0).

In layman's terms this means the following:
There are three actions that can be taken, each available at every state of the system except the final states.
Action `G` continues to operate the system, with a chance of failure. Continuing operations is always free. If the system is already out of control it will always stay out of control when continuing operations.
Action `H` will always reset the system back to in-control before continuing with operations. Resetting the system always has a cost but more so when the system is out of control. The age of the sensor is not affected by this. The transition of action `H` is as if action `G` is taken from state (0, t).
Action `J` resets the system and the sensors before continuing with operations. Resetting the system and the sensors always has a high cost but more

so when the system is out of control. The transition of action `J` is as if action `G` is taken from state (0,0).
These actions are labeled as the letters in the blue boxes in figure 3.1.

3.2.3 Observations

The following describes observations of the POMDP:

- At each transition epoch the model perform a costless imperfect observation O = {0,...,y}.

It is important to note that only a single observation is made at every state. This observation gives some indication of what state the system is in. How these observation relate to the state is explained in the subsection on the observation function.

3.2.4 Transition function

The following describes the transition function of the POMDP:

- The system changes as a discrete time Markov chain over the dispositions. Since the actions do not have different transitional probabilities the current and next state are sufficient information to differentiate the transitional probabilities. Furthermore, the time is always incremented by 1 meaning information about the time is also unnecessary. In the model this has the notation $\mathrm{P}_{ij} = [0, 1]$ with $i, j \in D$. The transitional probability P_{ij} then means the the probability for state (i, t) to transition to state $(j, t + 1)$.

The transitional probabilities are displayed in figure 3.1 as the blue letters and numbers without an outline.

3.2.5 Observation function

The following description is given of the observation function

- The observational probabilities at state $(i, t) \in D \times \mathbb{N}$ can be denoted as a set Q(i,t) = $\{ q(i, t, 0), q(i, t, 1), .., q(i, t, y-1), q(i, t, y)\}$ with $q(i, t, k)$ denoting the individual observational probability for $k \in O$

- The function for determining singular observational probabilities is the probability mass function for independent trials in a binomial distributions. For this instances this is defined as follows: $q(i, t, k)$ is defined as $q(i, t, k) = \binom{y}{k} p_i(t)^k (1 - p_i(t))^{y-k}$ for $i \in D, k \in O, |O| = y$. Furthermore $p_0(t) = a \cdot t + b$ and $p_1(t) = -a \cdot t + c$ with $a, b, c \in [0, 1]$ and $b <= c$. Finally $\forall t, p_i(t) \in [0, 1]$

The observation function explains for every possible observation the chance it will be observed in a certain state. For each state the result of the observation functions over all observations sum up to exactly 1, corresponding to that only 1 actual observation will be made.
The more unique part of this model is that the observations are affected by how long the system has been operating. This is why t is a variable in the observation function. Depending on the exact definition of $q(i, t, k)$ this can have a number of different affects.
If q(i,t,k) = q(i,t',k) for $\forall t, t' \in T$ then the observations are not affected by time. This happens when in $p_i(t)$, $a = 0$. This means that no deterioration takes place and removes what makes the model of interest.
Additionally the definitions of $p_0(t)$ and $p_1(t)$ should be chosen in such a way that $p_0(n) = p_1(n)$ with n being the maximum value of T. This will make it more difficult to distinguish what state the system is in as time increases until it is completely impossible to distinguish.
Since it can be quite difficult to visualize this, it may help to look at figure 3.4 in the next section. There you can see how in the numerical example presented in section 3.4, as the age of the sensor increases the chance that an observation can be made in both the in control and out of control state increases as well.
In figure 3.1 the observation function is displayed as the function in the red box attached to the red arrow. It is only displayed for the two observations on display.

3.2.6 Reward function

The following description is given of rewards in this model:

- C_d is the cost for transition to an out of control state (1,t).
- C_s is the cost for full inspection. This cost is always incurred when taking action H.
- C_i is the cost for replacing the sensors. This cost is always incurred when taking action J.
- C_r is the cost for resetting the system. This cost is always incurred when action J or action H is taken from an out of control state (1,t)

Although still called reward, in this specific model it the reward function is used more as cost. Therefore maximizing rewards is probably not something that will be of interest with this model. The rewards are shown in the model inside the yellow rectangles.

3.3 Visual model of the POMDP

Using the description the POMDP, a visual depiction of the model was created. Figure 3.2 shows a smaller version over the period {0,1,2} and figure 3.3 shows a slightly larger version over the period {0,1,2,3,4,5}. As the description suggest, in principal the model can be extended over any period size. After the first state (0,0), it constitutes a constantly repeating pattern. The only exception are the final two states at the maximum time interval. In the case of figure 3.2 these are (0,2) and (1,2). From these final states a transition is no longer possible.

Figure 3.2: POMDP for (i, t) $\in \{0, 1\} \times \{0, 1, 2\}$

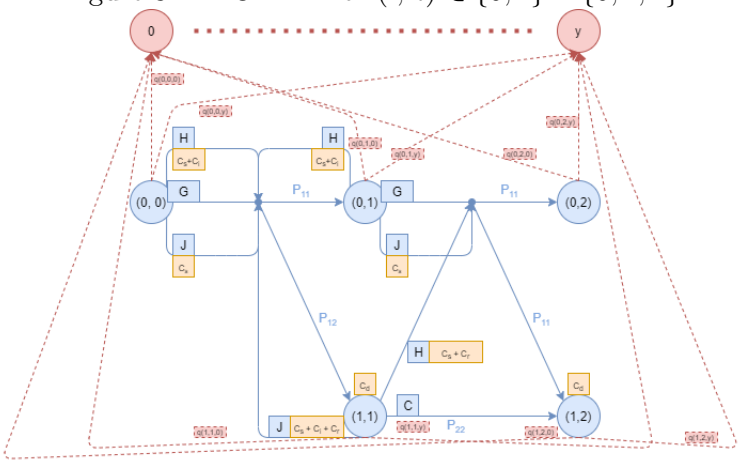
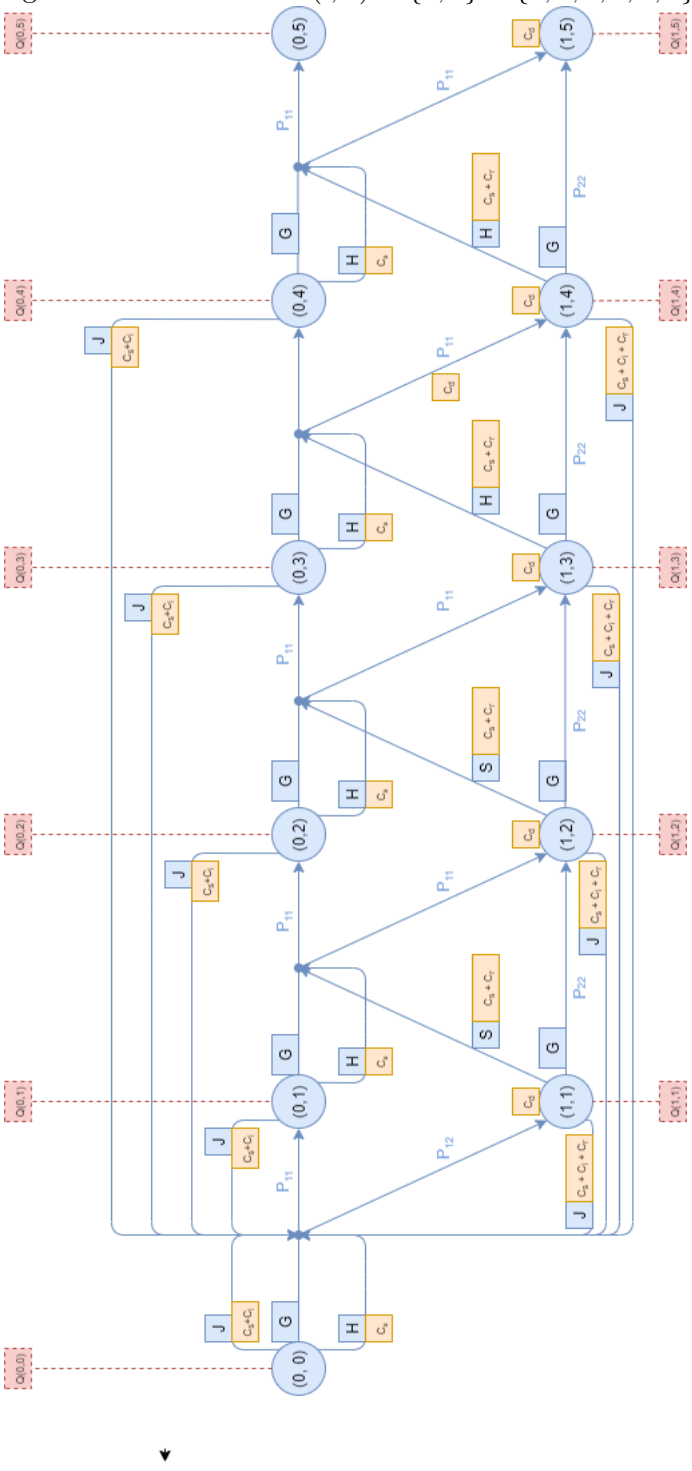


Figure 3.3: POMDP for (i, t) $\in \{0, 1\} \times \{0, 1, 2, 3, 4, 5\}$



3.4 Numerical example

The description above gives a generic POMDP whose observations are influenced by the age of the sensors. To implement this in PRISM a move from this generic implementation to a numerical implementation is necessary. For this, the first numerical example presented in [30] was used. There are two small changes from the model described in the paper. The state space was changed from $(i,t) \in \{0,1\} \times \{0,....,10\}$ to $(i,t) \in \{0,1\} \times \{0,..,5\}$. Furthermore, the rate of sensor deterioration was doubled to match the smaller state space. In the PRISM model different sizes where tested, so this change in size mostly applies to the description and depiction presented in this section. With these changes included, this implementation of the POMDP is defined as follows:

- The state space is $(i,t) \in \{0,1\} \times \{0,..,5\}$.
- The observations O = {0,...,50}, this means y=50.
- The transition matrix is $P_{11} = 0.9$, P_{12}=0.1, P_{21}=0 and P_{22}=0.
- Elements of Q(i,t) are given by $q(i,t,k) = \binom{y}{k} p_i(t)^k (1-p_i(t))^{y-k}$ for $i \in D, k \in O, p_1(t) = 0.4+0.030t$ and $p_2(t) = 0.7-0.030t$. This implies that for t >= 5 sensor decisions are completely uninformative. This means that for t = 5 the probability of observing o ∈ O is the same regardless of whether the observation is made from the in-control or out-of-control state.
- C_d=100, C_s=75, C_r=50 and C_i=20.

Using python to calculate the values of the observations, figure 3.4 showcases the probability mass function at different sensor ages. Figure 3.5 shows the model with all the values filled in.

Figure 3.4: Probability mass function of the sensor observations as a function of the sensor age. Blue is the in control state "0", red is the out of control state "1"

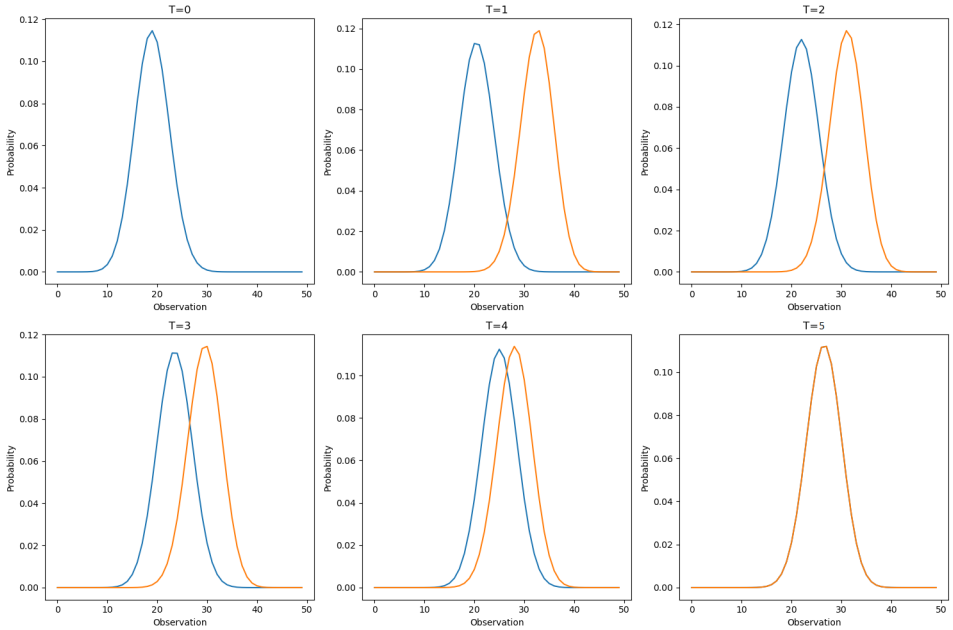
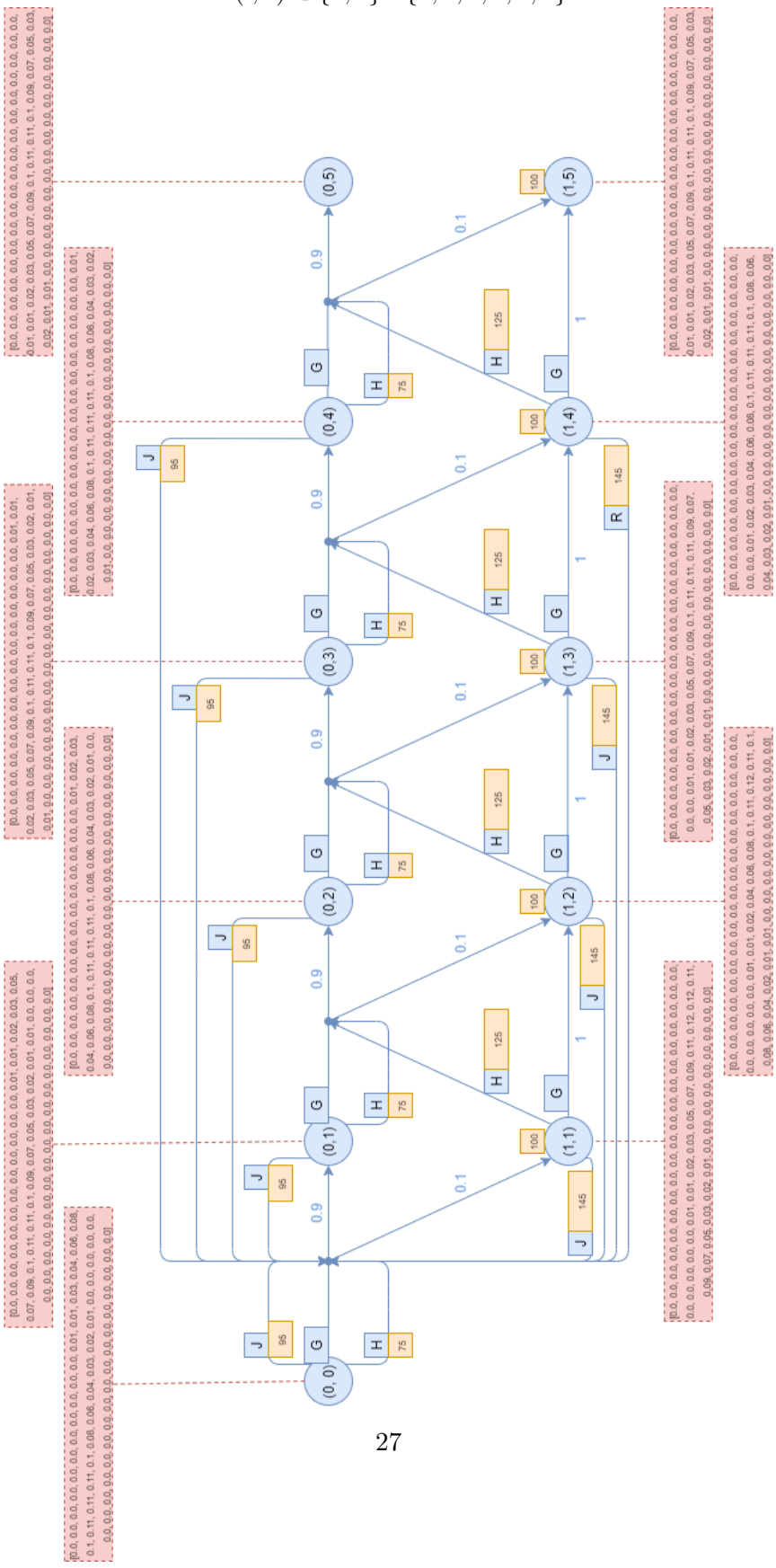


Figure 3.5: POMDP for (i, t) $\in \{0, 1\} \times \{0, 1, 2, 3, 4, 5\}$ with numerical values



3.5 Modeling the MDP in PRISM

PRISM was used to develop a model of the underlying MDP from the POMDP. When first working on the problem PRISM did not yet support POMDPs, so it was decided to first focus on the MDP. Seeing as this is only a MDP both observations and a observation function are not part of the model. Furthermore, since action J only really makes sense if the age of the model affects the observations, action J was left out of the model. This led to the following generic description of the PRISM model:

```
mdp
// constants
const double p11 = transitional probability from in control to
    in control;
const double p12 = transitional probability from in control to
    out of control;
const double p22 = transitional probability from out of control
    to out of control;

const int cs = cost of action H, guaranteeing return to in
    control state;
const int cr = additional cost of action h if performed from out
     of control state;
const int cd = cost of operating in out of control state;
const int maxt = maxtmum time interval;

//modules
module m1
        i : [0..1] init 0;
        t : [0..maxt] init 0;

        //from in-control
        [G] i = 0 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);
        [H] i = 0 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);

        //from out-of-control
        [G] i = 1 & t < maxt -> p22:(i'=1) & (t'=t+1);
        [H] i = 1 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);

endmodule
label "end"= t = maxt;

rewards
        i = 1: cd;
        [H] i = 0: cs;
        [H] i = 1: cs+cr;
endrewards
```

A few things to note with the model:

- A description for the variables is given to give a generic version of this model. A numerical version based on the example discussed earlier can be found in appendix.
- The value i = 0 is used for in control and the value i = 1 is used for out of control in the model.
- The constant "maxt" is used to indicate the maximum value "t" can have within this model.
- The actions are labelled with what action they correspond too. This is used in the reward function to apply action specific costs to action `H`.
- The cost of operating in the out-of-control disposition is applied to the state and not the transition.
- Technically, this implementation creates the state (1,0) which should not exits. This state is unreachable however, and it was not removed for simplicity reasons.

3.6 Modeling the POMDP in PRISM

After having developed the MDP attention was shifted to making the POMDP. This of course is a lot more interesting since the unique identifying feature of this model is related to the observation function. After much consideration the following generic implementation of the model was developed:

pomdp

```
observables
        o, t, b
endobservables

// constants
const double p11 = transitional probability from in control to
    in control;
const double p12 = transitional probability from in control to
    out of control;
const double p22 = transitional probability from out of control
    to out of control;

const int ci = additional cost of action J, resetting t to 0;
const int cs = cost of action H, guaranteeing return to in
    control state;
const int cr = additional cost of action H and j if performed
    from out of control state;
const int cd = cost of operating in out of control state;
const int maxt = size of the time interval;
const int max0 = size of the observation space;
```

```
module m1
        i : [0..1] init 0;
        t : [0..maxt] init 0;
        o : [0..maxo] init 0;
        b : bool init true;

        //t=0
        [o] i=0 & t=0 & !b -> probability of observing o = 0 for
             t = 0 and i=0:(o' = 0) & (b' = true) + ..... +
            probability of observing o = maxo for t = 0 and i=0:(
            o' = max0) & (b' = true);
        [o] i=0 & t=0 & !b -> probability of observing o = 0 for
             t = 0 and i=1:(o' = 0) & (b' = true) + ..... +
            probability of observing o = maxo for t = 0 and i=1:(
            o' = max0) & (b' = true);
        .....
        //t=maxt
        [o] i=0 & t = maxt & !b -> probability of observing o =
            0 for t = maxt and i=0:(o' = 0) & (b' = true) + .....
             + probability of observing o = maxo for t = maxt and
             i=0:(o' = max0) & (b' = true);
        [o] i=0 & t = maxt & !b -> probability of observing o =
            0 for t = maxt and i=1:(o' = 0) & (b' = true) + .....
             + probability of observing o = maxo for t = maxt and
             i=1:(o' = max0) & (b' = true);



        //from in-control
        //G
        [G] i = 0 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
            false) + p12:(i'=1) & (t'=t+1) & (b'=false);
        //H
        [H] i = 0 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
            false) + p12:(i'=1) & (t'=t+1) & (b'=false);
        //J
        [J] i = 0 & t < maxt & b -> p11:(i'=0) & (t'= 1) & (b'=
            false) + p12:(i'=1) & (t'= 1) & (b'=false);

        //from out-of-control
        //G
        [G] i = 1 & t < maxt & b -> p22:(i'=1) & (t'=t+1) & (b'=
            false);
        //H
        [H] i = 1 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
            false) + p12:(i'=1) & (t'=t+1) & (b'=false);
        //J
        [J] i = 1 & t < maxt & b -> p11:(i'=0) & (t'= 1) & (b'=
            false) + p12:(i'=1) & (t'= 1) & (b'=false);

endmodule
label "end"= t = maxt;
```

```
label "incontrol" = i = 0;
label "outofcontrol" = i = 1;

rewards
        i = 1: cd;
        [H] i = 0: cs;
        [H] i = 1: cs+cr;
        [J] i = 0: cs+ci;
        [J] i = 1: cs+ci+cr;
endrewards
```

Considerably more interesting design choices were made in this model compared to the MDP. In the following subsections these will be explained. This again is a generic implementation of the model. An implementation that models a numerical example of section 3.4 is added in the appendix. Other examples are also available on the Github repository.

3.6.1 Bool b and guards

The design choice was made to distinguish between states where observations are made and those where the actions take place with a bool b. This would have been avoided if at all possible, since it increases the size of the model. Normally you would use multiple modules and through labels synchronise the observations with the actions of the model. This would require the actions and the observations to have the same labels. This however is not possible. The reason for this is that the actions have the same guard, and sharing a label is then not allowed when in the same module. It is also not possible to have these actions in different modules since that would mean they synchronise with each other as well as with the observations. The guards in the model are therefore not used for synchronisation but mostly for clarity. It is necessary for the guards of the observations to all be the same because the POMDP functionality of PRISM will generate an error if it is not possible to take the same action from different states. For all these reasons it was a necessary design decision to add this guard. The labels of action G, H and J also relate to the reward.

3.6.2 Observation

Perhaps the most interesting and most distinguishing feature is the way the observations are set up. PRISM does not allow for complex functions. This is problematic because typically the observational probabilities for this model are a binomial distribution, beyond the capabilities of PRISM to turn into a function. Furthermore it is not allowed to use variables as part of transitional probabilities. This leaves as only solution to generate all observational probabilities for any given period size and size of observational space beforehand. Therefore, to the authors knowledge, it is not possible to

generalize this model using variables within PRISM. Instead different specific implementations need to be made for differing sizes of observational space and periods. To this end a simple script was developed in Java that allows you to create a custom observation function where you can pick your own observational and period size. Furthermore the definition of this function is according to the specifications given in 3.2.5.

3.7 Properties of the PRISM model

Having developed the model, now properties can be used to model check the POMDP. Three properties will be discussed here.

3.7.1 Reachability

The first property tests whether it is possible to reach the final state of the model. The property is:

```
P>=1 [ F "end" ]
```

This property means "The algorithm eventually reaches the end label with probability 1". The end label is given to states that have the maximum value of t. If this value is true it indicates that termination is always possible.

3.7.2 Probability of reaching the end

The second property tests what the probablity is of reaching the end of the model if it is our intention to do so. the property is:

```
Pmax=? [ F "end" ]
```

This property mean "The maximum expected probability of the algorithm eventually reaching end". The reason to include this property is also explained in the results, but has to do with some of the model checkers being unable to process the reachability property.

3.7.3 Expected minimum cost

The thrid property test what the cost is that is expected to incur when we try to minimize the cost. The property is:

```
Rmin=? [ F "end" ]
```

This property means "The minimum expected reward if the algorithm eventually reaches end". Note that expected minimum value does not mean the minimum value but the minimum value taking transitional probabilities into account. This is by far the most interesting and relevant property to check if you plan on using the model for optimization.

3.8 Results with varying observation space

The following sections shows the results achieved when model checking the properties from the previous section. The model was checked for three different sizes of the observations space. This was chosen as the only variable since it allowed the rest of the model to stay much more consistent. For the observation space, a maxo of 1, 5 and 10 was chosen. Although the different solution methods indicate a different number of states at different sizes of the model, the model should at least grow by the size of the state space for each observation added, and the number of transitions should increase by threefold that. This made it impossible to perform property verification on models with a larger observation space for this state space. When keeping all other variables of this model equal, 8 observations is when in most model checkers I started experiencing problems. At 20 observations all models were unable to perform property verification. All values besides the observations space correspond with the numerical example described in section 3.4.

3.8.1 Reachability

Table 3.1: Reachability

maxo	P ≥ 1 [F"end"]		
	Storm	Prism	Toolchain
1	not possible	false	1
5	not possible	false	1
10	not possible	false	1

Table 3.1 displays the results of testing the reachability property, P ≥ 1 [F"end"]. Lets cover the results for each of the outcomes separately.

Storm As specified in the documentation for properties in Storm [4] it is not possible to verify a property in a non-deterministic model without knowing if it needs to be minimized or maximized. As a workaround the property "Pmax=? [F "end"]" was also attempted. This property does in fact work and for all sizes that were attempted it provided an answer of 1.

Prism For Prism the reachability property is false. However, this is not actually the result of the property verification but is more so the result of only min and max properties being defined for POMDPs in Prism [2]. Just like with Storm we can attempt to alleviate this by using the property "Pmax=? [F "end"]. For this property Prism gives a result of ≈ 0.5. This is an average of the bounds it finds for this property however. The bounds it

finds are 0.0 and ≈1.0. These respectively represent an under-approximation and an over-approximation of the given property [24]. These bounds and how to interpret them will be further discussed in section 3.9.1.

Toolchain The Toolchain is capable of model checking this property in an appropriate manner to show that in fact the "end" is reachable from the initial state.

3.8.2 Expected minimum cost

Table 3.2: Expected minimum cost

maxi	Rmin=? [F"end"]		
	Storm	Prism	Toolchain
1	[146.42, 178.316734]	177.011682292056	186.385936625735
5	[141.2648347, 180.135328]	166.215066356857	174.488417923098
10	not possible	157.49152121347	not possible

Depending on what solution method used the answer for this property takes a slightly different format. For Storm a range of answers is delivered. Prism provides an answer with a margin for error. All the margins for error were less then one in this case, and therefore left out to make it more legible. The Toolchain provides an exact solution but cannot guarantee a minimal value. The Toolchain is designed to provide guarantees on a threshold, but this does not guarantee optimal value. All the solution methods are not perfect and therefore this should not be seen as an exact solution. They do give a good estimation however. Since they all also fall within the same range of solutions one can be fairly sure that these give a good indication of what the expected minimum cost can be if an good policy is applied. There also seems to be a correlation between the amount of available observations and the overall costs. This is to be expected since with maxo = 1 only 2 observations are available. Even in states with a low value for t this makes the observations barely indicative of the true state of the system. With a higher amount of observations they become much more disjoint in indicating the state of the system, especially for low values of t. Finally it should be noted that for a maxo = 10 both Storm and the Toolchain seem unable to solve the property. The best explanation that can be given for this is that it seems to create numerical errors for larger sizes of the model as currently designed since it results in an 8 decimal probability. Since the Toolchain uses the Storm python API it makes sense that if Storm fails, it fails too. To resolve this adaptations to the program generating observations would have to be made.

3.9 Results with varying period sizes

The model was also tested in the model testing environments for different period sizes. During these test the number of observations was set at 5. Furthermore as little alterations as possible were made to the numerical implementation used previously. In order for the deterioration to match the new sizes of the model a slight alteration was made to the observation function. This alteration simply means that the distinguishing between in control and out of control states becomes impossible at the new maximum interval instead of earlier. For the period size 2 this means that in the observation function for P_i a=0.75 was used, and for period size 10 a=0.015.

3.9.1 Probability of reaching the end

Table 3.3: Probability of reaching the end

maxt	Pmax=? [F"end"]		
	Storm	Prism	Toolchain
2	1	0.4999999995	1
5	1	0.499999996355	1
10	1	0.5000000002952447	1

In the previous section we have already seen that reachability property does not work in Storm and Prism. Therefore instead the probability of reaching the end was also briefly discussed. Here the results of that property are displayed since they did produce some interesting results in Prism. First of, both in Storm and in the tool chain the property produced results much as expected and in line with the results we saw in the previous section.

Now the results in Prism. As discussed in the previous section the number Prism outputs is simply the average of a lower and an upper bound. The lower bound is always 0.0. This means the higher bound is always twice the average that is displayed in the table. Furthermore Prism always indicates how far off it can be from the actual solution. In both this and the previous section the output could be offset both positively and negatively by the number Prism gives as a final answer. Now for this model one would expect the higher bound to be exactly 1.0, since always taking action `G` should always result in the model reaching the label "end". This is because even when out of control the label "end" is simply the maximum period. However for smaller periods this upper bound is slightly less then that and for larger periods it is slightly more. This same seemed to happen when increasing the size of the observation space. For both observation size

10 and period size 10 this upper bound even exceeds 1. At first I thought this was simply an oddity but after looking carefully through my models I found one size for which this did not happen. This was when the period was 5 and max0 1. The explanation for this seems to be the following: For most models, when generating the observational probabilities, the sum of all probabilities from a certain state does not perfectly round to 1 due to rounding errors when generating the individual probabilities. It is however always incredibly close to 1 usually being of by just a ten thousandths or less. This is close enough for the model checkers to not see this as a reason for generating an error about the probabilities not summing up to 1. At maxO 1 the probabilities do however sum to exactly 1. This seems to indicate that although Prism still accepts a model for probabilities that are not perfectly 1, it still works with these faulty probabilities and is able to generate an upper limit higher than 1. This was a little surprising for me since these probabilities are also not related to the transitions, just to the observations. This is however the only explanation I could find. Although for me this was a surprising find, this is however not a wrong answer because the final solution including bound it gives falls within the range of a correct solution.

3.9.2 Expected minimum cost

Table 3.4: Expected minimum cost

maxt	Rmin=? [F"end"]		
	Storm	Prism	Toolchain
1	0	0	1.5078488488788224
2	[19.9999999, 19.9999999]	20.000000236113443	23.12232870713059
5	[141.2648347, 180.135328]	166.215066356857	174.488417923098
10	not possible	458.2009471674771	not possible

The expected minimum costs for larger period sizes does not produce many interesting results. As with a large number of observations Storm and the Toolchain get numerical errors for a large period size. As expected the cost is significantly lower for lower operation times. The one result that I found quite interesting is that for a period size of 1 both Storm and Prism expect no cost, unlike the Toolchain which accounts for a little amount of cost. Since there is a state based reward related to being out of control, and there is a chance that you transition immediately to out of control, one would expect a small amount of expected cost. Since with this period size after transition the end of the model is reached immediately, this means that it seems that Storm and Prism check if the final state is reached before applying

any state based cost or reward on the final state, while the Toolchain does apply a state based cost before checking if a final state is reached.

Chapter 4

Related Work

Optimal policies for safety critical systems and its deteriorating sensor [30] provides a description for the model used in this thesis. They also provide some conditions for an optimal policy concluding with the assessment that there is always a threshold structure in the policy until which it is advantageous to generally take actions that do not reset the sensors and after which it always is advantageous to take actions that do reset the sensor. The contributions added to this by this thesis are primarily turning this description in a generalized PRISM model and performing model checking on instances presented in the paper. Furthermore the presentation of the model in this thesis is more fleshed out and descriptive making it a lot more accessible to a layman.

Robust finite-state controllers for uncertain pomdps [9] presents the toolchain developed at the Radboud university that was used in this thesis for Model checking. This paper is a much more general paper about uncertain POMDPs and how this toolchain is shown to be able to solve these problems witch decent benchmarks and good results. In this thesis the toolchain is specifically applied to the problem of a system with deteriorating sensors.

Chapter 5

Conclusions

In this thesis safety critical system with deteriorating sensors was successfully modeled as a POMDP in the PRISM language. A visual model of a POMDP was developed for a system with deteriorating observations. A generalized PRISM model was created that allows one to create a specific implementation with the use of number of variables available to the practitioner. A small variety of possible implementations were shown within the thesis and different properties were model checked accordingly. Relatively consistent reward properties were obtained when testing the model in different model checking environments.

Bibliography

[1] Prism. http://www.prismmodelchecker.org/.

[2] Prism language manual version 4.7. https://www.prismmodelchecker.org/manual/, Mar 2021.

[3] Storm. https://www.stormchecker.org, 2021.

[4] Storm documentation. `https://www.stormchecker.org/documentation/background/properties.html`, 2021.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[6] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[7] Alexander Bork, Sebastian Junges, Joost-Pieter Katoen, and Tim Quatmann. Verification of indefinite-horizon pomdps. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 288–304, Cham, 2020. Springer International Publishing.

[8] Darius Braziunas. Pomdp solution methods. *University of Toronto*, 2003.

[9] Murat Cubuktepe, Nils Jansen, Sebastian Junges, Ahmadreza Marandi, Marnix Suilen, and Ufuk Topcu. Robust finite-state controllers for uncertain pomdps. *CoRR*, abs/2009.11459, 2020.

[10] Bob Givan and Ron Parr. An introduction to markov decision processes. *Purdue University*, 2001.

[11] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 comparison of tools for the analysis of quantitative formal models. In Dirk Beyer, Marieke

Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 69–92, Cham, 2019. Springer International Publishing.

[12] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reability. Technical report, 1990.

[13] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *CoRR*, abs/2002.07080, 2020.

[14] Highfill Jannett and Michael Mcasey. An optimal control problem in economics. *International Journal of Mathematics and Mathematical Sciences*, 14, 01 1991.

[15] Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Parameter synthesis for markov models. *CoRR*, abs/1903.07993, 2019.

[16] Sebastian Junges, NH Jansen, Ralf Wimmer, Tim Quatmann, Leonore Winterer, J-P Katoen, and Bernd Becker. Finite-state controllers of pomdps via parameter synthesis. 2018.

[17] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[18] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 547–550, New York, NY, USA, 2002. Association for Computing Machinery.

[19] Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[20] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[21] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1-2):5–34, 2003.

[22] PG Morato, KG Papakonstantinou, CP Andriotis, JS Nielsen, and P Rigo. Optimal inspection and maintenance planning for deteriorating structures through dynamic bayesian networks and markov decision processes. *arXiv preprint arXiv:2009.04547*, 2020.

[23] Jannie Sønderkær Nielsen and John Dalsgaard Sørensen. Maintenance optimization for offshore wind turbines using pomdp. In A. Der Kiureghian and A. Hajian, editors, *Reliability and Optimization of Structural Systems*, pages 175–182. American University of Armenia Press, Yrevan, Armenia, 2012. Reliability and Optimization of Structural System : IFIP WG 7.5 working group conference, IFIP WG 7.5 2012 ; Conference date: 24-06-2012 Through 27-06-2012.

[24] Gethin Norman, David Parker, and Xueyi Zou. Verification and control of partially observable probabilistic systems. *Real-Time Systems*, 53(3):354–402, 2017.

[25] R.W.H. Sargent. Optimal control. *Journal of Computational and Applied Mathematics*, 124(1):361–371, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[26] Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339. Citeseer, 1997.

[27] Johan Sijtsma, Nils H Jansen, and Frits W Vaandrager. Creating a formal model of the game 2048. 2020.

[28] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088, 1973.

[29] Matthijs T. J. Spaan. *Partially Observable Markov Decision Processes*, pages 387–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] Chiel van Oosterom, Lisa M. Maillart, and Jeffrey P. Kharoufeh. Optimal maintenance policies for a safety-critical system and its deteriorating sensor. *Naval Research Logistics (NRL)*, 64(5):399–417, 2017.

[31] Yu-Fen Zhang, Qun-Feng Zhang, and Rui-Hua Yu. Markov property of markov chains and its test. In *2010 International Conference on Machine Learning and Cybernetics*, volume 4, pages 1864–1867, 2010.

[32] Karl Johan Åström. Optimal control of markov processes with incomplete state information i. 10:174–205, 1965.

Appendix A

Appendix

This appendix contains the Java program that can be used to generate your own observations for a model further specified by you. Furthermore it contains a Numerical implementation of the MDP that was developed and a numerical implementation of the POMDP that was developed. All these things, the other POMDPs used in the results section and the Toolchain used in modelchecking can also be found on the following github: https://github.com/vkoenv/bachelor_thesis.

A.1 Program for generating observation space

The following java program can be used to generate the observational transitions necessary for any size of the model.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.text.DecimalFormat;

public class ObservableMaker {
    int sizeo;
    int tmax;
    double a0;
    double a1;
    double b0;
    double b1;
    private static DecimalFormat df = new DecimalFormat("
        0.00000000");
    /**
     * Constructor for the observable maker classes setting up
         the variables
     * @param sizeo size of observation space
     * @param tmax size of time interval
     * @param a0 coefficient for p1 = a0*t + b0
     * @param a1 coefficient for p1 = a1*t + b1
     * @param b0 constant for p1 = a0*t + b0
```

```
     * @param b1 constant for p1 = a1*t + b1
     */
    public ObservableMaker(int sizeo, int tmax, double a0,
        double a1, double b0, double b1){
        this.sizeo = sizeo;
        this.tmax = tmax;
        this.a0 = a0;
        this.a1 = a1;
        this.b0 = b0;
        this.b1 = b1;
    }
    /**
     * creates factorial n
     * @param n size of the factorial
     * @return !n
     */
    public BigDecimal factorial(int n){
        if(n>0)
            return new BigDecimal(n).multiply(factorial(n-1));
        else
            return new BigDecimal(1);
    }
    /**
     * Creates a combination for any n and r
     * @param n total number of objects in the set
     * @param r number of choosing objects from the set
     * @return number of combinations
     */
    public BigDecimal combi(int n, int r){
        return factorial(n).divide((factorial(n-r).multiply(
            factorial(r))));
    }
    /**
     * Generates the transitional probability of a single
         observation at a specific time given the state of the
         system.
     * @param k the observation
     * @param s0 the state of the system, true is in-control,
         false is out-of-control
     * @param t the time
     * @return the transitional probability for observation k at
         time t and state s0.
     */
    public BigDecimal probability(int k, boolean s0, int t){
        if(s0)
            return combi(sizeo,k).multiply(new BigDecimal(a0*t+
                b0).pow(k)).multiply(new BigDecimal(1-(a0*t+b0)).
                pow(sizeo-k));
        else
            return combi(sizeo,k).multiply(new BigDecimal(a1*t+
                b1).pow(k)).multiply(new BigDecimal(1-(a1*t+b1)).
                pow(sizeo-k));
    }
    /**
```

```
     * Generates a binomial distribution
     * @param s0 the state of the system, true is in-control,
         false is out-of-control
     * @param t the time
     * @return the transitional probabilities at time t and
         state s0
     */
    public BigDecimal[] Binom(boolean s0,int t){
        BigDecimal[] bigD = new BigDecimal[sizeo+1];
        for(int k = 0; k<=sizeo; k++) bigD[k] = probability(k,
            s0, t).round(new MathContext(8, RoundingMode.
            HALF_EVEN));
        return bigD;
    }
    /**
     * formats the binomial distribution for use in prism
     * @param bd the binomial distribution
     * @return a string for the update part of an observation
         command
     */
    public String Binomial( BigDecimal[] bd){
        String tussen = "";
        for(int i = 0; i<=sizeo; i++){
            if(i != 0)
                tussen += " + ";
            tussen += df.format(bd[i]) +  ":(o' = " + i + ") & (
                b' = true)";
        }
        tussen += ";";
        return tussen;
    }
    /**
     * formats the entire observation
     * @return formatted observations for use in prism
     */
    @Override
    public String toString(){
        String tussen = "";
        for(int t = 0; t<=tmax;t++){
            tussen+="//t=" + t;
            tussen += "\n[o] i=0 & t=" + t + "& !b -> " ;
            tussen += Binomial(Binom(true,t));
            tussen += "\n[o] i=1 & t=" + t + "& !b -> " ;
            tussen += Binomial(Binom(false,t));
            tussen += "\n";
        }
        return tussen;
    }
}

public class Main {

    public static void main(String[] args) {
```

```
        // fill in your value for size of the observation space
            (sizeo), maximum time (tmax), and the other variables
             that make up the bionomial coefficient.
        ObservableMaker observableMaker = new ObservableMaker(5,
            5, 0.03, 0.03, 0.4, 0.7);
        System.out.println(observableMaker.toString());
    }
}
```

A.2 Numerical implementation of MDP

The following is a single numerical implementation of the MDP. It model the underlying MDP for a specific model.

```
mdp
// constants
const double p11 = 0.9;
const double p12 = 0.1;
const double p22 = 1;

const int ci = 20;
const int cs = 75;
const int cr = 50;
const int cd = 100;
const int maxt = 5;

//modules
module m1
        i : [0..1] init 0;
        t : [0..maxt] init 0;

        //from in-control
        [G] i = 0 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);
        [G] i = 0 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);

        //from out-of-control
        [H] i = 1 & t < maxt -> p22:(i'=1) & (t'=t+1);
        [H] i = 1 & t < maxt -> p11:(i'=0) & (t'=t+1) + p12:(i
            '=1) & (t'=t+1);

endmodule
label "end"= t = maxt;

rewards
        i = 1: cd;
        [H] i = 0: cs;
        [H] i = 1: cs+cr;
endrewards
```

A.3 Numerical implementation of POMDP

The following is a numerical impementaiton of one of the possible POMDPs. Specifically it models the POMDP in the results section for maxo = 5 and maxt = 5.

pomdp

```
observables
        o, t, b
endobservables


const double p11 = 0.9;
const double p12 = 0.1;
const double p22 = 1;

const int ci = 20;
const int cs = 75;
const int cr = 50;
const int cd = 100;
const int maxt = 5;
const int maxo = 5;

module m1
        i : [0..1] init 0;
        t : [0..maxt] init 0;
        o : [0..maxo] init 0;
        b : bool init true;

        //t=0
        [o] i=0 & t=0& !b -> 0.07776000:(o' = 0) & (b' = true) +
             0.25920000:(o' = 1) & (b' = true) + 0.34560000:(o' =
             2) & (b' = true) + 0.23040000:(o' = 3) & (b' = true)
             + 0.07680000:(o' = 4) & (b' = true) + 0.01024000:(o'
             = 5) & (b' = true);
        [o] i=1 & t=0& !b -> 0.00243000:(o' = 0) & (b' = true) +
             0.02835000:(o' = 1) & (b' = true) + 0.13230000:(o' =
             2) & (b' = true) + 0.30870000:(o' = 3) & (b' = true)
             + 0.36015000:(o' = 4) & (b' = true) + 0.16807000:(o'
             = 5) & (b' = true);
        //t=1
        [o] i=0 & t=1& !b -> 0.06016921:(o' = 0) & (b' = true) +
             0.22695402:(o' = 1) & (b' = true) + 0.34242186:(o' =
             2) & (b' = true) + 0.25831824:(o' = 3) & (b' = true)
             + 0.09743583:(o' = 4) & (b' = true) + 0.01470084:(o'
             = 5) & (b' = true);
        [o] i=1 & t=1& !b -> 0.00391354:(o' = 0) & (b' = true) +
             0.03972835:(o' = 1) & (b' = true) + 0.16132119:(o' =
             2) & (b' = true) + 0.32753091:(o' = 3) & (b' = true)
             + 0.33249350:(o' = 4) & (b' = true) + 0.13501251:(o'
             = 5) & (b' = true);
        //t=2
```

```
[o] i=0 & t=2& !b -> 0.04591650:(o' = 0) & (b' = true) +
     0.19557029:(o' = 1) & (b' = true) + 0.33319382:(o' =
     2) & (b' = true) + 0.28383178:(o' = 3) & (b' = true)
     + 0.12089131:(o' = 4) & (b' = true) + 0.02059630:(o'
     = 5) & (b' = true);
[o] i=1 & t=2& !b -> 0.00604662:(o' = 0) & (b' = true) +
     0.05374771:(o' = 1) & (b' = true) + 0.19110298:(o' =
     2) & (b' = true) + 0.33973862:(o' = 3) & (b' = true)
     + 0.30198989:(o' = 4) & (b' = true) + 0.10737418:(o'
     = 5) & (b' = true);
//t=3
[o] i=0 & t=3& !b -> 0.03450252:(o' = 0) & (b' = true) +
     0.16574742:(o' = 1) & (b' = true) + 0.31849505:(o' =
     2) & (b' = true) + 0.30600505:(o' = 3) & (b' = true)
     + 0.14700243:(o' = 4) & (b' = true) + 0.02824752:(o'
     = 5) & (b' = true);
[o] i=1 & t=3& !b -> 0.00902242:(o' = 0) & (b' = true) +
     0.07055995:(o' = 1) & (b' = true) + 0.22072600:(o' =
     2) & (b' = true) + 0.34523810:(o' = 3) & (b' = true)
     + 0.26999390:(o' = 4) & (b' = true) + 0.08445963:(o'
     = 5) & (b' = true);
//t=4
[o] i=0 & t=4& !b -> 0.02548040:(o' = 0) & (b' = true) +
     0.13801882:(o' = 1) & (b' = true) + 0.29904077:(o' =
     2) & (b' = true) + 0.32396083:(o' = 3) & (b' = true)
     + 0.17547878:(o' = 4) & (b' = true) + 0.03802040:(o'
     = 5) & (b' = true);
[o] i=1 & t=4& !b -> 0.01306912:(o' = 0) & (b' = true) +
     0.09023918:(o' = 1) & (b' = true) + 0.24923203:(o' =
     2) & (b' = true) + 0.34417757:(o' = 3) & (b' = true)
     + 0.23764642:(o' = 4) & (b' = true) + 0.06563568:(o'
     = 5) & (b' = true);
//t=5
[o] i=0 & t=5& !b -> 0.01845281:(o' = 0) & (b' = true) +
     0.11276719:(o' = 1) & (b' = true) + 0.27565312:(o' =
     2) & (b' = true) + 0.33690938:(o' = 3) & (b' = true)
     + 0.20588906:(o' = 4) & (b' = true) + 0.05032844:(o'
     = 5) & (b' = true);
[o] i=1 & t=5& !b -> 0.01845281:(o' = 0) & (b' = true) +
     0.11276719:(o' = 1) & (b' = true) + 0.27565313:(o' =
     2) & (b' = true) + 0.33690937:(o' = 3) & (b' = true)
     + 0.20588906:(o' = 4) & (b' = true) + 0.05032844:(o'
     = 5) & (b' = true);

//G
[g] i = 0 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
    false) + p12:(i'=1) & (t'=t+1) & (b'=false);
//H
[H] i = 0 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
    false) + p12:(i'=1) & (t'=t+1) & (b'=false);
//J
[J] i = 0 & t < maxt & b -> p11:(i'=0) & (t'= 1) & (b'=
    false) + p12:(i'=1) & (t'= 1) & (b'=false);
```

```
        //G
        [g] i = 1 & t < maxt & b -> p22:(i'=1) & (t'=t+1) & (b'=
            false);
        //H
        [H] i = 1 & t < maxt & b -> p11:(i'=0) & (t'=t+1) & (b'=
            false) + p12:(i'=1) & (t'=t+1) & (b'=false);
        //J
        [J] i = 1 & t < maxt & b -> p11:(i'=0) & (t'= 1) & (b'=
            false) + p12:(i'=1) & (t'= 1) & (b'=false);

endmodule
label "end"= t = maxt;
label "incontrol" = i = 0;
label "outofcontrol" = i = 1;

rewards
        i = 1: cd;
        [H] i = 0: cs;
        [H] i = 1: cs+cr;
        [J] i = 0: cs+ci;
        [J] i = 1: cs+ci+cr;
endrewards
```