Model checking rail networks

Benjamin Bordais, Thomas Mari, Julie Parreaux

Abstract

Rail systems are particularly sensitive to unexpected delays or random events. They are often equipped with timetables that describe the expected behavior of the system. To stick to this timetable, rail systems are equipped with regulation policies that give instructions in real time when changes occur in the traffic. In this report, we consider how tools such as model checkers can help in the design and evaluation of regulation policies. Usually, these tools work with formal models of rail system. Therefore, our purpose is to properly model a rail system and to analyze and generate regulation policies on this model.

Keywords : Rail system; Model checking; Abstract model.

I. Introduction

Rail systems are particularly sensitive to unexpected delays or random events. They are often equipped with timetables that describe the expected behavior of the system. To stick to this timetable, rail systems are equipped with regulation policies that give instructions in real time when changes occur in the traffic. For such a policy to be efficient, it has to operate in real time. Therefore, the regulation policy needs to precisely determine how each train has to react in any possible situation. Once a policy is designed, it is important to assess its performance, that is its efficiency at enforcing timetable realization. Such a verification is too complex to be done by human. One of the techniques that could be used is model checking. Basically, given a mathematical model and a logical formula describing the requirements, model checking checks if the model satisfies the formula. In this work, we will consider verification of models describing rail systems.

We want to model a real rail system. In this system, some delays may occur at each movement of the trains. These events are unpredictable, and conveniently represented through probabilities. That is why, in our model, we need to have some probabilities. Moreover, trains can accelerate or decelerate. These possibilities need to be taken into account using nondeterminism. A popular model that contains both probabilities and nondeterminism is Markov Decision Process [**?**] (MDP for short). Rail systems will be modeled as MDPs. It is also a discrete model, that means that, in our model, we will need to discretize time and space.

A real rail system ensures safety measures independently of any regulation policy. Hence, a model of a rail system needs to take this aspect into account [**?**]. Typically, the fact that it is impossible for two trains to collide is a necessary condition and that impossibility needs to appear in the model. That is only when safety measures are guaranteed, that we can study regulation policies.

A regulation policy consists in making trains accelerate or slow down according to the current state and the events that already occurred in the system. So, once we have a model of the system, we may want to test the efficiency of a policy. Another possibility to ensure the efficiency of the implemented regulation policy is to generate it automatically from the system specification so that it meets the desired guarantees.

Once a regulation policy is designed (whether it is automatically generated or defined by ourselves), we need to ensure that this policy respects some desired properties. For example, if a train is delayed, with a high probability, at some point in the future it will be on time. The most suitable logic to express this kind of properties is the Probabilistic Computational Tree Logic [**?**] [**?**] (PCTL in short).

The automatic tools we use for verification are model checkers. The most adequate model checkers to work on MDP and PCTL logic are PRISM [**?**] and Storm [**?**]. However, as any model checker, PRISM and Storm can only handle models of reasonable size. That is why we often have to abstract our modelization [**?**] [**?**] of the rail system. In counterpart, with an abstraction, we may lose in precision.

Our objective is to evaluate the performance of regulation policies on a real rail system. First, we need to build a model of the rail system. Then, once the correctness of the model is established (that is, the safety measures are respected), we can study a regulation policy, automatically generated or that we have defined ourselves. To do that, we will need model checkers, PRISM and Storm, to prove that the desired properties are enforced. For model checkers to scale correctly, we may have to abstract our model to simplify it.

This report is organized as follows : Section 2 provides a review of the notion of Markov Decision Process. Section 3 introduces our approach of the problem and the abstraction we consider. We conclude in Section 4.

II. State of the art

A. MDPs : Markov Decision Process

We introduce the notion of Discrete-Time Markov Chain which can be seen as a directed graph, in which vertices represent the state of the system, and the transitions represent the probability to change states.

Definition 1. *A Discrete-Time Markov Chain (DTMC) is a Tuple* $\mathcal{D} = (S, \bar{s}, P, AP, L)$ *where*

- S *is a finite non-empty set of states,*
- $\bar{s}$ *is an initial state*
- $P : S \times S \to [0,1]$ *is a transition probability matrix such that* $\forall s \in S, \sum_{s' \in S} P(s,s') = 1$
- AP *is a set of atomic propositions*
- $L : S \to 2^{AP}$ *is a labeling function such that* $\forall s \in S, L(s)$ *is the subset of* AP *assigned to* s.

Figure 1 represents a DTMC, the probabilities are given by the matrix P. $P(i,j)$ is the probability to move to state s_j when the system is in state s_i. These probabilities can be read directly on the vertices.

We introduce the notion of Markov Decision Process (MDP), which allows to represent the nondeterministic evolution of a system. In a given state, an agent can choose an action which yields a distribution of probabilities for the next transition.

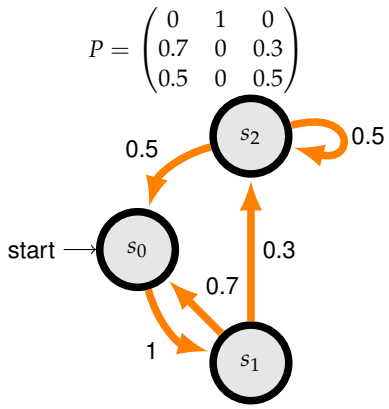

Figure 1. The DTMC $(\{s_0, s_1, s_2\}, s_0, P, \{a\}, L)$ with $L(s_0) = \{a\}$ and $L(s_1) = L(s_2) = \emptyset$

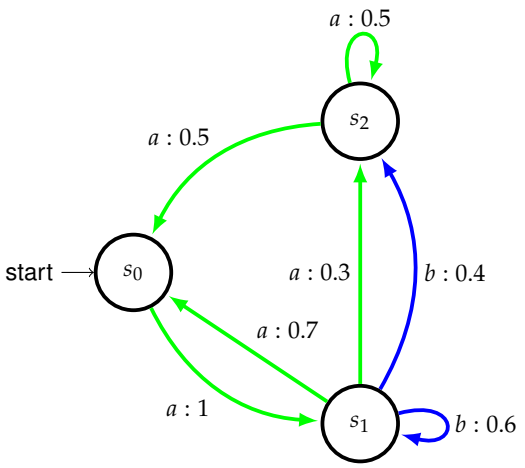

Figure 2. The MDP $(\{s_0, s_1, s_2\}, s_0, \{a, b\}, \delta, \{a\}, L)$ with $L(s_0) = \{a\}$ and $L(s_1) = L(s_2) = \emptyset$ avec $\delta(s, a, s') = P(s, s')$ (from the previous DTMC) and $\delta(s_1, b, s_1) = 0.6$, $\delta(s_1, b, s_2) = 0.4$

Definition 2. *A Markov Decision Process (MDP) is a tuple* $\mathcal{M} = (S, \bar{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, AP, L)$ *where*

- S, $\bar{s}$, AP *and* L *are as for DTMCs.*
- $\alpha_{\mathcal{M}}$ *a finite set of actions*
- $\delta_{\mathcal{M}} : S \times \alpha_{\mathcal{M}} \times S \to [0, 1]$ *where* $\delta_{\mathcal{M}}(s, \alpha, s')$ *represents the probability of going into the state* s' *from the state* s *when action* α *is chosen. In addition, we have that for* $s \in S$, *and* $\alpha \in \alpha_{\mathcal{M}}$, $\sum_{s' \in S} \delta_{\mathcal{M}}(s, \alpha, s') = 1$.

The actions are chosen nondeterministically. A policy allows to resolve the nondeterminism. At each state, a policy gives the action to take according to states visited and the actions taken so far.

Definition 3. *We define* $Path^{\mathcal{M}}$ *as the set of all sequences* $s_1 \alpha_1 s_2 \alpha_2 ...$ *such that* $\forall i \geq 0$ $s_i \in S, \alpha_i \in \alpha_{\mathcal{M}}$ *and* $P(s_i, \alpha, s_{i+1}) > 0$. $Path^{\mathcal{M}}_{fin}$ *is the set of finite prefixes of* $Path^{\mathcal{M}}$ *which terminate by a state. A policy is a function in* $Path^{\mathcal{M}}_{fin} \to \alpha_{\mathcal{M}}$

From an MDP $\mathcal{M}$ and a policy σ we can generate a DTMC $\mathcal{M}^{\sigma}$. However, it is not straightforward and we only present here the case where the policy is memoryless. That is, a policy σ such that $\forall m, m' \in Path^{\mathcal{M}}_{fin}$ with $last(m) = last(m')$ we have $\sigma(m) = \sigma(m')$ (the choice of the action to take only depends on the current state). We denote by $\sigma(s)$ the decision taken by σ when $s = last(m)$. Formally, with such a policy σ and an MDP $\mathcal{M} = (S, \bar{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, AP, L)$ we obtain the DTMC $\mathcal{M}^{\sigma} = (S, \bar{s}, P_{\sigma}, AP, L)$ with $P_{\sigma}(s, s') = \delta(s, \sigma(s), s')$. For example, the DTMC in Figure 1 can be generated from the MDP in Figure 2 and the (memoryless) policy σ such that $\forall m \in Path^{\mathcal{M}}_{fin}, \sigma(m) = a$.

B. Properties

Once our model is designed, we want to check whether the system satisfies some properties. One can distinguish situations that should always occur, and conversely situations that shall be avoided. Therefore properties are split into two categories. On the one hand, there are properties that express that a good event will happen (they are called *liveness* properties). For instance, the property "if a train has some delay, it will catch up this delay eventually" is a liveness property. On the other hand, there are properties that express that a bad event will not happen (they are called *safety* properties). The property "two trains will not collide" is a an example of a *safety* property.

It has to be noted that, since our model is probabilistic, some properties we want to check on our model also need to be probabilistic. So, it is interesting to have answers that are not Boolean. That is why we want to be able to weigh our properties by probabilities.

Now, we need a logic to express these properties. In this logic, we need to be able to express temporal properties (like the fact that an event may or may not happen). Moreover, it has to allow probabilistic properties. That is why we use the Probabilistic Computational Tree Logic (PCTL in short) [**?**], a probabilistic variant of the branching-time temporal logic CTL (Computational Tree-Logic) [**?**]. First, we have temporal connectors. For example, $F\ \phi$ expresses that at some point in the future ϕ is true. In the same way, $F^{\leq n}\ \phi$ means that within n steps, ϕ will be true. We also have the formula $X\ \phi$ which means that, at the next step, ϕ is true. On top of that, the PCTL logic allows to express that the probability that a formula holds is lower or higher than a given value. This is expressed by properties of the form $\phi = P_{\alpha\ p}[\psi]$ where $\alpha \in \{<, \leq, >, \geq\}$ and $p \in [0; 1]$. $P_{\alpha\ p}$ is called the external probabilistic connector in the formula ϕ.

For example, the safety property "two trains will not collide" is equivalent to "the probability of two trains colliding is equal to 0". Let ϕ be a formula expressing that two trains are colliding then the PCTL formula expressing the above property is $P_{\leq 0}[F\ \phi]$. Here is another example. The property "a train makes up its delay within 10 steps with a high probability" could be described by the PCTL formula $P_{\geq 0.9}[F^{\leq 10}\ \phi]$ if ϕ expresses that a train has no delay.

Given a PCTL formula, we want to check whether it holds in our model. Let us first consider a DTMC $\mathcal{D}$ and a PCTL formula without external probabilistic connector, typically $\psi = F\ \phi$. We define $P^{\mathcal{D}}(\psi)$ as the probability that ψ holds in the DMTC $\mathcal{D}$. Let us now consider a PCTL formula, for instance $\Phi' = P_{\leq p}[F\ \Phi]$. The first step consists in determining the set of states $\mathcal{S}_{\Phi}$ of the DMTC where Φ is true. Once this is done, computing the probability that $F\ \Phi$ holds consists in computing the probability of reaching the set of states $\mathcal{S}_{\Phi}$ from an initial state. That computation can be automated. Then Φ' is true for the DTMC $\mathcal{D}$ if $P^{\mathcal{D}}(F\ \Phi) \leq p$. This method can be applied to any PCTL formula. That is, determining if a PCTL formula holds in a DTMC can be reduced to the computation of the probability of reaching a set of states.

For example, we implemented the DTMC of Figure 1 in the model-checker PRISM. This allowed us to compute the probability :

$$P(F^{\leq 10}(state = s2)) \simeq 0.832$$

Hence, the PCTL formula $P_{\geq 0.8}(F^{\leq 10}(state = s2))$ holds in this DTMC, however $P_{\geq 0.9}(F^{\leq 10}(state = s2))$ does not.

Let us now focus on MDPs. We cannot use the method described earlier because the probability of reaching a set of states is not yet defined due to nondeterminism. So, we first need to define a policy and obtain a DTMC such that the probability of interest becomes computable. However, according to the policy we choose, we may not obtain the same result. That is why we need to define a minimum and maximum probability on MDP satifying a PCTL formula ϕ without external probability connector.

Definition 4. *Let $\mathcal{M}$ be an MDP and ϕ a PCTL formula without external probability connector. We define :*

$$P^{\mathcal{M}}_{min}(\psi) = min_{\sigma \in Policy} P^{\mathcal{M}^\sigma}(\psi)$$

$$P^{\mathcal{M}}_{max}(\psi) = max_{\sigma \in Policy} P^{\mathcal{M}^\sigma}(\psi)$$

where Policy is the set of all policies of the MDP $\mathcal{M}$.

Hence, in a PCTL formula for an MDP, each connector $P_{\alpha\ p}$ has to be changed into either $P_{min\ \alpha\ p}$ or $P_{max\ \alpha\ p}$. Once this change is done, the satisfiability of a PCTL formula for an MDP is analogous to the satisfiability for a DTMC.

We implemented the MDP of the figure 2 in PRSIM [**?**] to compute the minimum and maximum probability for the same property as before :

$$P_{min}(F^{\leq 10}(state = s2)) \simeq 0.831$$

$$P_{max}(F^{\leq 10}(state = s2)) \simeq 0.989$$

From this result, we can deduce that, for example, the formula $P_{min \geq 0.8}(F^{\leq 10}(state = s2))$ holds for the MDP whereas $P_{max \leq 0.9}(F^{\leq 10}(state = s2))$ does not.

The model checkers we use are PRISM[1] [**?**] and Storm[2] [**?**]. PRISM is easy to use and quite efficient at proving probabilistic properties on MDP and DTMC. However, it cannot deal with conditional. That is, we cannot express a property of the form $P_{\leq 0.5}(\phi_1)$ if the system remains in path satisfying ϕ_2. For instance, if we want to know the average number of steps necessary to reach a given set of states, we do not want to take into account the executions where this set of states is never reached. PRISM cannot be used in that case. That is why we also use Storm which is able to compute conditional probabilities.

Once a policy is designed, model checkers are able to check that some properties hold with that policy. However, designing a policy that fulfills our expectations may be tricky. Hopefully, we can also use model checkers to automatically generate a policy that ensures some properties. However, we do not know yet how to generate a policy with both of these tools.

C. Abstraction

When the size of the model is too big, model checkers are not powerful enough to verify properties. To reduce the size of the model, we use an abstraction of MDP [**?**] [**?**]. An abstraction of an MDP merges some states of the MDP to reduce their number. We present two methods of abstraction:

- three-valued abstraction;
- game-based abstraction.

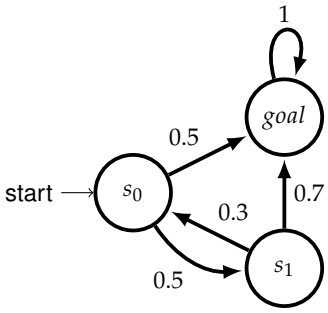

Figure 3. A very simple MDP with only one action available at each state

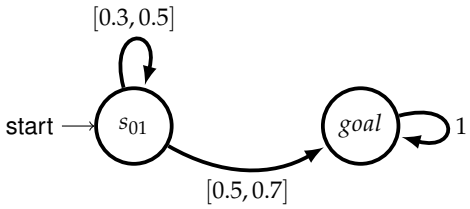

Figure 4. The abstraction of the MDP from figure 3

Three-valued abstraction: The three-valued abstraction partitions the states of an MDP into several classes $\mathcal{S}_1 ... \mathcal{S}_n$ and gathers some transitions whose origin or destination states are in the same set $\mathcal{S}_i$. Gathered transitions are abstract transitions. Different probabilities may occur on transitions that were gathered. That is why an abstract transition cannot be annotated by a probability but by an interval between a minimum and a maximum probability. We obtain an abstract MDP whose states are $\mathcal{S}_1 ... \mathcal{S}_n$. For instance, the MDP from Figure 3 can be abstracted into the MDP from Figure 4.

Then, the truth value of properties is not true or false. For instance, we know that formula $P_{\leq 0.6}(X\ goal)$ holds in the MDP of Figure 3, however, we do not know if it holds in its abstraction in Figure 4. That is why the truth value can be true, false or unknown (hence the three-valued abstraction). When the truth value of a property is true or false, we can conclude on the original model. That is, the property holds (resp. does not hold on the original model). However, when the truth value is unknown we cannot conclude.

Game-based abstraction: Let us consider the previous abstraction and the resulting abstract MDP. If we consider a state of the abstract MDP, we do not know exactly to which state of the original MDP it corresponds. Thus, our abstraction introduced another nondeterminism to the already existing nondeterminism of the original MDP. So, the nondeterminism of the abstract MDP is due to both the nondeterminism of the original model and the nondeterminism of the abstraction. The game-based abstraction consists in resolving the two kind of abstractions by two different players. Player one resolves the nondeterminism of the abstraction and player two resolves the nondeterminism of the original model.

Depending on the role of each player, we may have a lower or a upper bound on the maximal or minimal reachability property. For instance, if both players try to minimize the reachability probability, we will have a lower bound on the minimal reachability probability. On the other hand, if one player tries to maximize the reachability probability and the other tries to minimize it, we might have a

[1] http://www.prismmodelchecker.org/
[2] http://www.stormchecker.org/

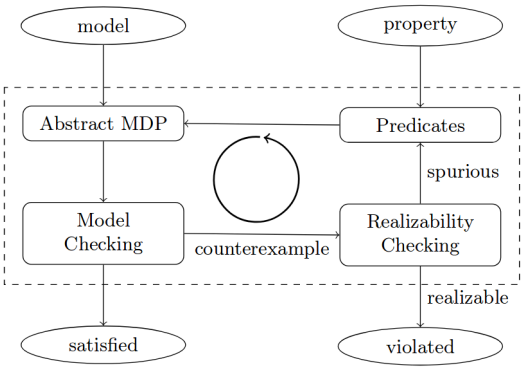

Figure 5. A schema of the CEGAR loop (figure taken from [**?**])

different, tighter approximation.

The lower and upper bounds obtained on the minimal and maximal reachability probability can be used to prove that a PCTL formula holds on the original model if the approximation is not too coarse.

Counter-example guided abstraction refinement : If an abstract model violates a property, it is possible for model checkers to produce a counterexample that witnesses that violation. If the counterexample is realizable in the original model, then it violates the property. However, if not, this counterexamples is spurious. That means that the abstraction is too coarse and that we have to refine it in a way that avoids this spurious counterexample. Then, we need to restart the verification process with the refined abstract model. This approach is called Counterexample-guided abstraction refinement (CEGAR). A diagram of this operation can be found at Figure 5. This method can be used with any abstraction technique to properly refine our abstraction.

III. Proposition of solution

On a real rail system, a train faces random events and may be delayed. These delays are not predictable, hence we need to use probabilities to correctly model the system. Moreover, trains can change their speed. That phenomenon is modeled by nondeterminism. For these reasons, we use MDP to model rail system.

MDP is a discrete model. Therefore we need to discretize time and space. In the real system, train moves are continuous. In the MDP, trains moves forward step by step, with every step corresponding to a fixed amount of time. We discretize the distance between two stations by adding intermediate points between these stations. The distance between two points is also fixed. In this model, the speed is inversely proportional to the time needed to travel between two stations.

To be realistic, this system must respect physical and security constraints. Thereby, the following constraints must be ensured by our model:

- Two trains must not collide.
- One train must not overtake another train.

These properties are independent of regulation policies and must always be verified by the model. For instance, in terms of PCTL formula, an MDP $\mathcal{M}$ that models a real system needs to ensure that the formula $P_{max\leq 0}(F\ collision)$ holds.

We begin by modeling a single rail line. We assume that this line is a ring without intersection to simplify the model. Indeed, intersections are more difficult to handle because we have to determine priorities on

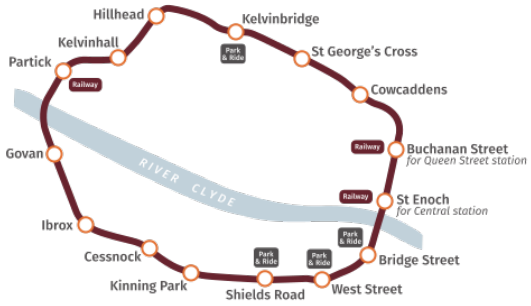

Figure 6. A schema of the subway line of Glasgow

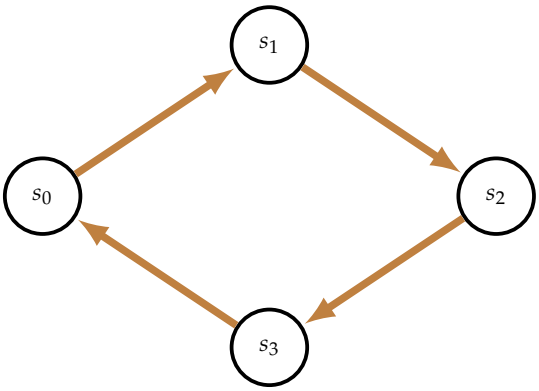

Figure 7. The simplified model we first want to study

trains that need to enter the intersection. On this ring, trains always move in the same direction. This topology is simpler and remains reasonable for a rail line. In fact, this model is quite accurate for the subway line of Glasgow, that can be seen at figure 6.

As mentioned earlier, we introduce intermediate points between stations to take the discretization of space into account. We call location either a point or a station. In this way, the distance between two stations is depicted by the number of intermediate points between two consecutive stations, given that the distance d between two locations is constant. At each step, trains may move forward from a location to another. The duration t of the steps fixes our discretization of time. The smaller the distance the more accurate the discretization.

In our model, trains can only stay at their current location or go to the next one at each step. That means that the maximum speed for train is $v_{max} = \frac{d}{t}$. To illustrate this model, let us consider the MDP of Figure 7 that is an abstraction of the real system of Figure 6 where we have only four stations.

The section between stations is depicted by the figure 8. It is obvious that a train cannot go from s_0 to s_1 in less than 3 time steps. That is another way of seeing the maximum speed. It is the ratio between the distance $s_0 - s_1$ (which is $3 \times d$) and the minimum time possible to cross that distance (which is $3 \times t$). Then, we have that $v_{max} = \frac{3\times d}{3\times t} = \frac{d}{t}$.

At each location, there are three possible actions. Action a yields the green transitions, whereas action b yields the blue transitions and action c the red ones. Note that all three actions are not necessarily always allowed. Indeed, this model needs to ensure safety measures. For examples, actions a and b are forbidden at each location where there is a train in the next location. Clearly enough this restriction ensures that trains do not collide as well as they do not overtake each other.

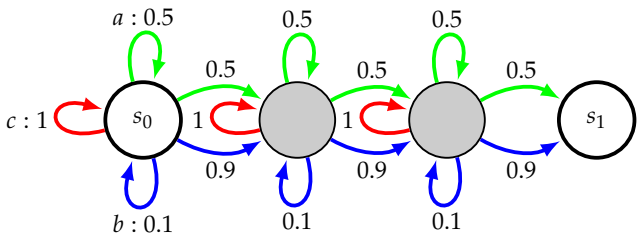

Figure 8. The modelization between two stations with two intermediate points

In this model it is easy to associate a speed to each action. By looking at the probability to go forward at each step, we note that action a corresponds to a medium speed, action b to a higher speed and action c to a stop.

To be more precise, we can compute the speed induced by choosing each action. For example, let us consider action b. The probability of going to the next location in one step is equal to $p_b = 0.9$. It is straightforward that the probability of going to the next location at step $k \geq 1$ is equal to $0.1^{k-1} \times 0.9$. Hence, that probability follows a geometric law. Therefore, the expected value of such a law is equal to $E_b = \frac{1}{p_b} = \frac{1}{0.9} = \frac{10}{9}$. That means that, on average, it will take E_b steps to go from a location to the next one. So, the average speed is equal to :

$$v_b = \frac{\text{distance between two succesives locations}}{\text{average time needed to go from a location to the next one}} = \frac{d}{E_b \times t} = \frac{d}{1/p_b \times t} = p_b \times \frac{d}{t} = \frac{9 \times d}{10 \times t}$$

In the same way, we can compute the average speed obtained by choosing action a. That is:

$$v_a = p_a \times \frac{d}{t} = \frac{d}{2 \times t}$$

Finally, we have that the speed for action c is $v_c = 0$. These results confirm the intuition we had by looking at probability of going forward.

In this model, we can design a very simple regulation policy. The idea is that action b stands for a higher speed than action a, thus action b has to be taken whenever a train is delayed. So, the policy is that whenever a train has some delays, it carries out action b, in other case it takes action a (assuming these actions are available, that is there is no train in the next location).

Now that we have a (quite simple) regulation policy, we would like to check its effectiveness. We need to define the formula ϕ stating that a train has some delay. Once it is done, we will be able to define the PCTL formulas of interest. For instance, we would like that the formula $\phi \Rightarrow P_{\geq 0.9}[F^{\leq 10}\ \neg\phi]$ holds in our model with this very simple regulation policy. That is, we would like that delays are recovered in less than ten steps with a high probability. If it does not, we may have to refine this regulation policy so that nicer properties hold.

IV. Conclusion

We study models of rail systems for regulation purposes. An appropriate model could allow us to design an effective regulation policy. That is, a policy that makes trains catch up their delays.

A suitable model is Markov Decisions Process, which take into account potential delays of trains while allowing to design a regulation policy. To assess the effectiveness of a regulation policy we can use model checkers.

The solution presented here is quite simple. It consists in discretizing time and space by adding intermediate points between stations. At each step, the action chosen determines the speed of the train. The regulation policy we first consider is to accelerate whenever a train has some delay.

We are going to see to what extent the policy considered here is efficient in terms of catching up delays by implementing the model in a model checker. We will likely need to study the accuracy of our discretization. Likewise, the probabilities on the transitions have to be adjusted to better picture the real system. The size of our model may constrain us to design an abstraction. We might have to design an abstraction specific to our case. Then, we will try to generate with model checkers some regulation policies. Finally, we will study a more complex topology of the system.

Acknowledgment

We want to thanks our supervisors Nathalie Bertrand, Loïc Hélouët and Ocan Sankur for their time and their help all along this semester.