DeepDECS Quick Tutorial

March 19, 2023

Using DeepDECS is straightforward, and in this short tutorial we will take you through the basics in under 15 minutes. In doing so, we will illustrate the application of DeepDECS to a hypothetical toy example in which an autonomous car assesses the condition of the next segment of road in front of it at regular time intervals, and decides to drive over it "fast" or "slow" depending on whether a DNN classifier predicts it to be icy or not.

1 Prepare the DeepDECS inputs

Prior to using DeepDECS, you need to prepare its four inputs (see Fig. 1a from our DeepDECS paper [1]):

1. the DNN classifier that the autonomous system uses to perceive its environment;
2. a representative test dataset for this DNN;
3. a perfect-perception discrete-time Markov chain (DTMC) modelling the behaviour of the system;
4. the system requirements encoded in probabilistic computation tree logic (PCTL).

1.1 DNN classifier

Applying DeepDECS requires a DNN classifier that distinguishes between the different environment states that the autonomous system needs to consider in its decisions. This perception DNN needs to be trained prior to using DeepDECS.

For the hypothetical autonomous car from our example, the perception DNN needs to be a two-class DNN classifier that takes sensor inputs about the state of the environment and predicts whether the road segment ahead is not covered in ice (class $k = 1$) or is icy (class $k = 2$).

1.2 Test dataset

To use DeepDECS, you will also need a large test dataset that is *representative* of the inputs of each class that the perception DNN will encounter when the autonomous system is deployed in its operational design domain. This test dataset will be used to quantify the uncertainty of your DNN classifier in the first DeepDECS stage.

For the hypothetical autonomous car and its two-class DNN classifier, we will suppose that this test dataset comprises $10,000$ data samples from non-icy road segments and an equal number of samples from icy road segments. In general, the test dataset does not need to include equal numbers of data samples of each

1.3 Perfect-perception DTMC model

DeepDECS also requires as input a DTMC that instantiates the DeepDECS *generic perfect-perception DTMC* from Fig. 2a in our paper [1] and models the behaviour of the autonomous systems under the (ideal) assumption that it is capable of perceiving the state of its environment with 100% accuracy at all times.

Listing 1 shows the instance of the DeepDECS *generic perfect-perception DTMC* that models the operation of our hypothetical autonomous car. The DTMC comprises four modules: (i) ManagedComponents, (ii)

```
1  dtmc
2
3  module ManagedComponents // autonomous car
4    z : [0..1] init 0;
5
6    [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
7    [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
8  endmodule
9
10 module EnvironmentMonitor
11   k : [1..2]; // k=1 - no ice; k=2 - ice
12
13   [monitor] t=2 -> 0.6:(k'=1) + 0.4:(k'=2);
14 endmodule
15
16 const double x1; //probability of not slowing down on non-icy road (k=1)
17 const double x2; //probability of not slowing down on icy road (k=2)
18
19 module Controller
20   slow : bool init false;
21
22   [decide] t=3 & k=1 -> x1:(slow'=false) + (1-x1):(slow'=true);
23   [decide] t=3 & k=2 -> x2:(slow'=false) + (1-x2):(slow'=true);
24 endmodule
25
26 module Turn
27   t : [1..3] init 1;
28
29   [driveFast] true -> 1.0:(t'=2);
30   [driveSlow] true -> 1.0:(t'=2);
31   [monitor]   true -> 1.0:(t'=3);
32   [decide]    true -> 1.0:(t'=1);
33 endmodule
34
35 rewards "risk"
36   [driveFast] k=2 : 1;
37 endrewards
38
39 rewards "distance"
40   [driveFast] true : 3;
41   [driveSlow] true : 1;
42 endrewards
```

Listing 1: DTMC for the perfect perception model in PRISM language

EnvironmentMonitor, (iii) Controller, and (iv) Turn. The ManagedComponents module models the car driving fast or slow, as specified by the Controller, which decides if the car should slow down ($slow' = true$) with probabilities $1 - x1$ and $1 - x2$ when a non-icy ($k' = 1$) or icy ($k' = 2$) road, respectively, was perceived by the EnvironmentMonitor. We note that the Controller is allowed to opt for fast driving over a road segment that is perceived to be icy for the situation in which the risk associated with this option is low, e.g., because the car is equipped with winter tires. Finally, the Turn module models the synchronisation between the ManagedComponents, EnvironmentMonitor and Controller modules. Our DTMC is augmented with two reward structures:

1. A *risk* reward structure (lines 35–37 from Listing 1) that associates a risk level of 1 with each icy road segment ($k = 2$) over which the car drives fast; by default, a risk level of 0 is associated to driving slow over icy road segments, and to driving (fast or slow) over non-icy road segments.

2. A *distance* reward structure (lines 39–42 from Listing 1) that associates a travelled distance of 3 with fast driving over a road segment, and a travel distance of 1 with slow driving over a road segment (i.e., we assume that the road condition is monitored at regular time intervals, and that road segments over which the car drives fast are three times longer than those over which the car drives slow).

1.4 PCTL-encoded requirements

The last input that you will need in order to use DeepDECS is a PCTL encoding of the requirements (constraints and optimisation objectives) for your autonomous system. The PCTL syntax, semantics and the

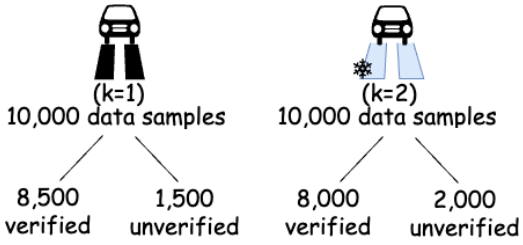


Figure 1: Partitioning the test dataset using a single DNN verification technique

use of PCTL to express such requirements are explained in our paper [1].

As an illustration, we will assume that the aim of the autonomous car from our example is to minimise the risk and to maximise the driven distance accrued over 1000 DTMC state transitions (corresponding to driving across 1000/3 road segments, since three state transitions are taken for each road segment as the turn variable t goes full cycle through its three values $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$). These two optimisation objectives are encoded in PCTL as:

$$\begin{array}{l} \mathsf{minimise}\ \mathcal{R}^{\mathsf{risk}}[C^{<=1000}] \\ \mathsf{maximise}\ \mathcal{R}^{\mathsf{distance}}[C^{<=1000}] \end{array} \tag{1}$$

where $\mathcal{R}^{\mathsf{risk}}[C^{<=1000}]$ and $\mathcal{R}^{\mathsf{distance}}[C^{<=1000}]$ represent the expected risk and distance cumulated over 1000 DTMC state transitions, respectively.

2 DeepDECS Stage 1: DNN uncertainty quantification

In this DeepDECS stage, you need to quantify the uncertainty introduced by the use of your DNN to perceive the environment of the autonomous system. This stage comprises the two steps detailed below.

Step 1: Test dataset partition. You can carry out the uncertainty quantification using zero, one, or several *DNN verification techniques*. As explained in our DeepDECS paper [1], where are couple of such techniques are presented, each of them takes as input your DNN and one data sample from the test dataset in Section 1.2, and outputs a boolean value v which indicates whether the DNN is likely to classify the data sample correctly ($v = \mathsf{true}$) or incorrectly ($v = \mathsf{false}$). As such, using n verification techniques will allow you to partition your test dataset into 2^n subsets of data samples: the subset containing data samples which all verification techniques indicated as unlikely to be classified correctly by the DNN, the subset containing sample that the first technique indicated as likely to be classified correctly but all the other techniques indicated as unlikely to be classified correctly, etc. This partition is not required if you use zero verification techniques ($2^0 = 1$).

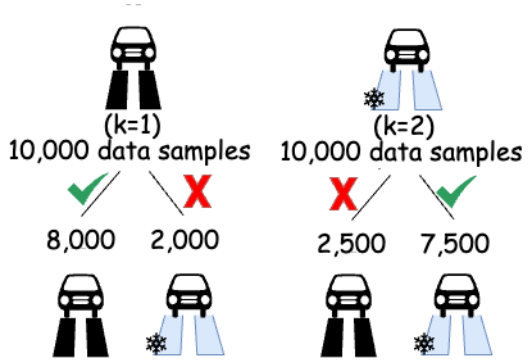
We illustrate the test dataset partition step in Fig. 1. This figure shows how the use of a single DNN verification technique can lead to a (hypothetical) partition of the $10,000$ data samples of non-icy road segments into $8,500$ "verified" data samples (i.e., data samples for which the verification techniques returns a true result) and $1,500$ "unverified" data samples; and to a similar partition of the $10,000$ data samples of icy road segments into $8,000$ "verified" and $2,000$ "unverified" samples.

Step 2: Confusion matrix generation. In this step, you need to use each of the subsets of test data samples from the Step 1 to obtain a separate confusion matrix for your DNN.

We illustrate this step for the autonomous car from our example in Fig. 2 (for a setup in which no DNN verification technique is used) and in Fig. 3 (for a setup in which the DNN verification technique that produced the test dataset partition from Fig. 1 was used).

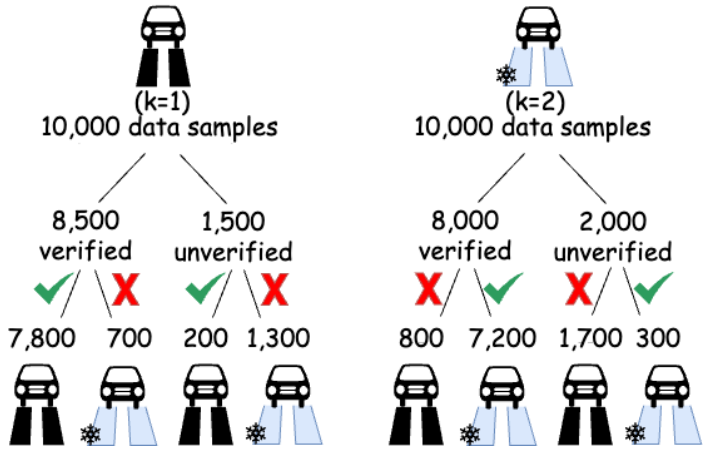
3 DeepDECS Stage 2: Model augmentation

This DeepDECS stage is fully automated, so you will only need to run the DeepDECS model augmentation tool. This tool takes three file names as command-line arguments. The first of these files must contain the perfect-perception DTMC model from Section 1.3, the second file must contain the rows of the confusion



$$\mathcal{C} = \begin{bmatrix} 8,000 & 2,000 \\ 2,500 & 7,500 \end{bmatrix}$$

Figure 2: Confusion matrix generation when no DNN verification technique is used: $8,000$ of the $10,000$ data samples from non-icy road segments, and $7,500$ of the $10,000$ data samples from icy road segments are classified correctly (left), yielding the confusion matrix on the right.



$$\mathcal{C}_{\mathsf{false}} = \begin{bmatrix} 200 & 1,300 \\ 1,700 & 300 \end{bmatrix}$$

$$\mathcal{C}_{\mathsf{true}} = \begin{bmatrix} 7,800 & 700 \\ 800 & 7,200 \end{bmatrix}$$

Figure 3: Confusion matrix generation when the DNN verification technique that produced the test dataset partition from Fig. 1 is used: Separate confusion matrices are obtained for the verified and for the unverified data samples. From the $1,500$ unverified data samples from non-icy road segments and $2,000$ unverified data samples from icy road segments, the DNN classifies correctly only 200 and 300 samples, respectively. In contrast, from the $8,500$ verified data samples from non-icy road segments and $8,000$ verified data samples from icy road segments, the DNN classifies correctly $7,800$ and $7,200$ samples, respectively. These results are summarised by the two confusion matrices on the right.

matrices from Section 2 (one row per line), and the third file is created by the tool and will contain the *DNN-perception DTMC model of the autonomous system.*

Assuming that the perfect-perception DTMC from Listing 1 is stored in a file named `car.pm` and the confusion matrix from Fig. 2 is used to create a file named `conf_mat.txt` and containing the lines:

```
8000 2000
2500 7500
```

the command

```
python deepDECSAugment.py car.pm conf_mat.txt car_with_DNN_no_verif.pm
```

produces a file named `car_with_DNN_no_verif.pm` that contains the DNN-perception autonomous car DTMC model from Listing 2. Similarly, the command

```
python deepDECSAugment.py car.pm conf_mat2.txt car_with_DNN_verif.pm
```

where the confusion matrix file `conf_mat.2txt` contains the rows of the two confusion matrices from Fig. 3 in

```
dtmc

module ManagedComponents // autonomous car
  z : [0..1] init 0;

  [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
  [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
endmodule

module EnvironmentMonitor
  k : [1..2] init 1; // k=1 - no ice; k=2 - ice
  k2: [1..2] init 1; // DNN perception of the real world

  [monitor] t=2 -> 0.6*(8000/10000):(k'=1)&(k2'=1) + 0.6*(2000/10000):(k'=1)&(k2'=2) +
                   0.4*(2500/10000):(k'=2)&(k2'=1) + 0.4*(7500/10000):(k'=2)&(k2'=2);
endmodule

const double x1; //probability of not slowing down on non-icy road (k=1)
const double x2; //probability of not slowing down on icy road (k=2)

module Controller
  slow : bool init false;

  [decide] t=3 & k2=1 -> x1:(slow'=false) + (1-x1):(slow'=true);
  [decide] t=3 & k2=2 -> x2:(slow'=false) + (1-x2):(slow'=true);
endmodule

module Turn
  t : [1..3] init 1;

  [driveFast] true -> 1.0:(t'=2);
  [driveSlow] true -> 1.0:(t'=2);
  [monitor]   true -> 1.0:(t'=3);
  [decide]    true -> 1.0:(t'=1);
endmodule

rewards "risk"
  [driveFast] k=2 : 1;
endrewards

rewards "distance"
  [driveFast] true : 3;
  [driveSlow] true : 1;
endrewards
```

Listing 2: DNN-perception autonomous system DTMC produced by the DeepDECS model augmentation tool for the setup with no DNN verification technique used in the uncertainty quantification stage

the format:

```
200 1300
1700 300
7800 700
800 7200
```

generates a file named `car_with_DNN_verif.pm` that contains the DNN-perception autonomous car DTMC model from Listing 3.

A description of the augmented DTMC models from Listings 2 and 3 is beyond the scope of this tutorial: to understand these models, we recommend that you refer to the DeepDECS paper [1]. However, we note that:

- The first model (Listing 2) uses a controller with two parameters:
  - x1, the probability of selecting fast driving when the DNN classifies the next road segment as non-icy (i.e., when $k2 = 1$);
  - x2, the probability of selecting fast driving when the DNN classifies the next road segment as icy (i.e., when $k2 = 2$).
- The second model (Listing 3) uses a controller with four parameters:

```
dtmc

module ManagedComponents
  z : [0..1] init 0;

  [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
  [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
endmodule

const double pVerif0WhenClass0 = 0.15;
const double pVerif0WhenClass1 = 0.2;
const double pVerif1WhenClass0 = 0.85;
const double pVerif1WhenClass1 = 0.8;

const double pClass0AsClass0Verif0 = 0.13333333333333333;
const double pClass0AsClass1Verif0 = 0.8666666666666667;
const double pClass1AsClass0Verif0 = 0.85;
const double pClass1AsClass1Verif0 = 0.15;

const double pClass0AsClass0Verif1 = 0.9176470588235294;
const double pClass0AsClass1Verif1 = 0.08235294117647059;
const double pClass1AsClass0Verif1 = 0.1;
const double pClass1AsClass1Verif1 = 0.9;

module EnvironmentMonitor
  k : [1..2]; // k=1 - no ice; k=2 - ice
  k2 : [1..2] init 1;
  v : [0..1] init 0;

  [monitor] t=2 -> 0.6*pVerif0WhenClass0*pClass0AsClass0Verif0:(k'=1)&(v'=0)&(k2'=1)+
                   0.6*pVerif0WhenClass0*pClass0AsClass1Verif0:(k'=1)&(v'=0)&(k2'=2)+
                   0.4*pVerif0WhenClass1*pClass1AsClass0Verif0:(k'=2)&(v'=0)&(k2'=1)+
                   0.4*pVerif0WhenClass1*pClass1AsClass1Verif0:(k'=2)&(v'=0)&(k2'=2)+
                   0.6*pVerif1WhenClass0*pClass0AsClass0Verif1:(k'=1)&(v'=1)&(k2'=1)+
                   0.6*pVerif1WhenClass0*pClass0AsClass1Verif1:(k'=1)&(v'=1)&(k2'=2)+
                   0.4*pVerif1WhenClass1*pClass1AsClass0Verif1:(k'=2)&(v'=1)&(k2'=1)+
                   0.4*pVerif1WhenClass1*pClass1AsClass1Verif1:(k'=2)&(v'=1)&(k2'=2);
endmodule

const double x1Verif0;
const double x1Verif1;
const double x2Verif0;
const double x2Verif1;

module Controller
  slow : bool init false;

  [decide] t=3 & k2=1 & v=0 ->  x1Verif0:(slow'=false) + (1-x1Verif0):(slow'=true);
  [decide] t=3 & k2=1 & v=1 ->  x1Verif1:(slow'=false) + (1-x1Verif1):(slow'=true);
  [decide] t=3 & k2=2 & v=0 ->  x2Verif0:(slow'=false) + (1-x2Verif0):(slow'=true);
  [decide] t=3 & k2=2 & v=1 ->  x2Verif1:(slow'=false) + (1-x2Verif1):(slow'=true);
endmodule

module Turn
  t : [1..3] init 1;

  [driveFast] true -> 1.0:(t'=2);
  [driveSlow] true -> 1.0:(t'=2);
  [monitor]   true -> 1.0:(t'=3);
  [decide]    true -> 1.0:(t'=1);
endmodule

rewards "risk"
  [driveFast] k=2 : 1;
endrewards

rewards "distance"
  [driveFast] true : 3;
  [driveSlow] true : 1;
endrewards
```

Listing 3: DNN-perception autonomous system DTMC produced by the DeepDECS model augmentation tool for the setup with one DNN verification technique used in the uncertainty quantification stage

- x1Verif0, the probability of selecting fast driving when the DNN classifies the next road segment as non-icy (i.e., when $k2 = 1$) and the DNN input used for this prediction is "unverified" (i.e., $v = 0$);
- x1Verif1, the probability of selecting fast driving when the DNN classifies the next road segment as non-icy (i.e., when $k2 = 1$) and the DNN input used for this prediction is "verified" (i.e., $v = 1$);
- x2Verif0, the probability of selecting fast driving when the DNN classifies the next road segment as icy (i.e., when $k2 = 2$) and the DNN input used for this prediction is "unverified" (i.e., $v = 0$);
- x2Verif1, the probability of selecting fast driving when the DNN classifies the next road segment as icy (i.e., when $k2 = 2$) and the DNN input used for this prediction is "verified" (i.e., $v = 1$).

4 DeepDECS Stage 3: Controller synthesis

In the final DeepDECS stage, you will need to use a probabilistic model checker such as PRISM [2] or Storm [3] to assemble a set of controllers that meet any constraints that your autonomous system requirements from Section 1.4 may include, and achieve Pareto-optimal trade-offs among the optimisation objectives from these requirements. Assuming that you are synthesising probabilistic controllers, you have two options. We describe each of these options in turn.

Controller synthesis option 1. The first option is to discretise the design space of the controller, e.g., by considering all values from the set $\{0.0, 0.1, \ldots, 0.9, 1.0\}$ for the probabilities with which the controller takes each of its actions. If you use this option, you have the opportunity to examine the outcome of using the controllers corresponding to all combinations of such probabilities, and to then identify those combinations that yield controllers which:

1. satisfy all constraints from your requirements for the autonomous system;
2. are Pareto-optimal with respect to the optimisation objectives from these requirements.

We will illustrate the use of this first option (in conjunction with the probabilistic model checker PRISM) for the autonomous car from our example, considering the setup that used a single DNN verification technique (i.e., the DNN-perception DTMC from Listing 3). Figs. 4 and 5 show how the PRISM model checker can be used to run an "experiment" that examines the expected *risk* for this DTMC when the values of the four controller parameters listed at the end of Section 3 are discretised and varied between 0 and 1 with step 0.1. The results of this experiment, and of an analogous experiment for the "distance" PCTL property need to be saved into separate comma-separate value (.csv) files (Fig. 6) for processing outside PRISM. Once the two .csv files are generated, we combine them into a single file that lists the expected risk and expected distance associated with each combination of parameter values. We display in Fig. 7a an excerpt from the table in this file, which can then be used to obtain a Pareto front for the two optimisation objectives from (1). Fig. 7b depicts (blue line) the Pareto front produced using this procedure and the R script from Listing 4, alongside the Pareto fronts obtained in the same way for the ideal-perception DTMC from Listing 1 (black line) and the DNN-perception DTMC with no verification from Listing 2 (red line). As expected, the (practically unattainable) perfect-perception front Pareto-dominates the DNN-perception front for the one verification technique setup, which in turn Pareto-dominates the DNN-perception front for the no verification technique setup.

Controller synthesis option 2. As a second option, you can use a search procedure to explore possible combinations of controller parameter values. This exploration can be performed using random search, or a more sophisticated search method such as multi-objective genetic algorithms (e.g., by using the EvoChecker tool for probabilistic model synthesis [4], which employs PRISM or Storm as a backend probabilistic model checking engine). Detailing this option is beyond the scope of our tutorial, so we refer you to the EvoChecker description [4] and website (`https://cs.york.ac.uk/tasp/EvoChecker`) for details on how to use it.

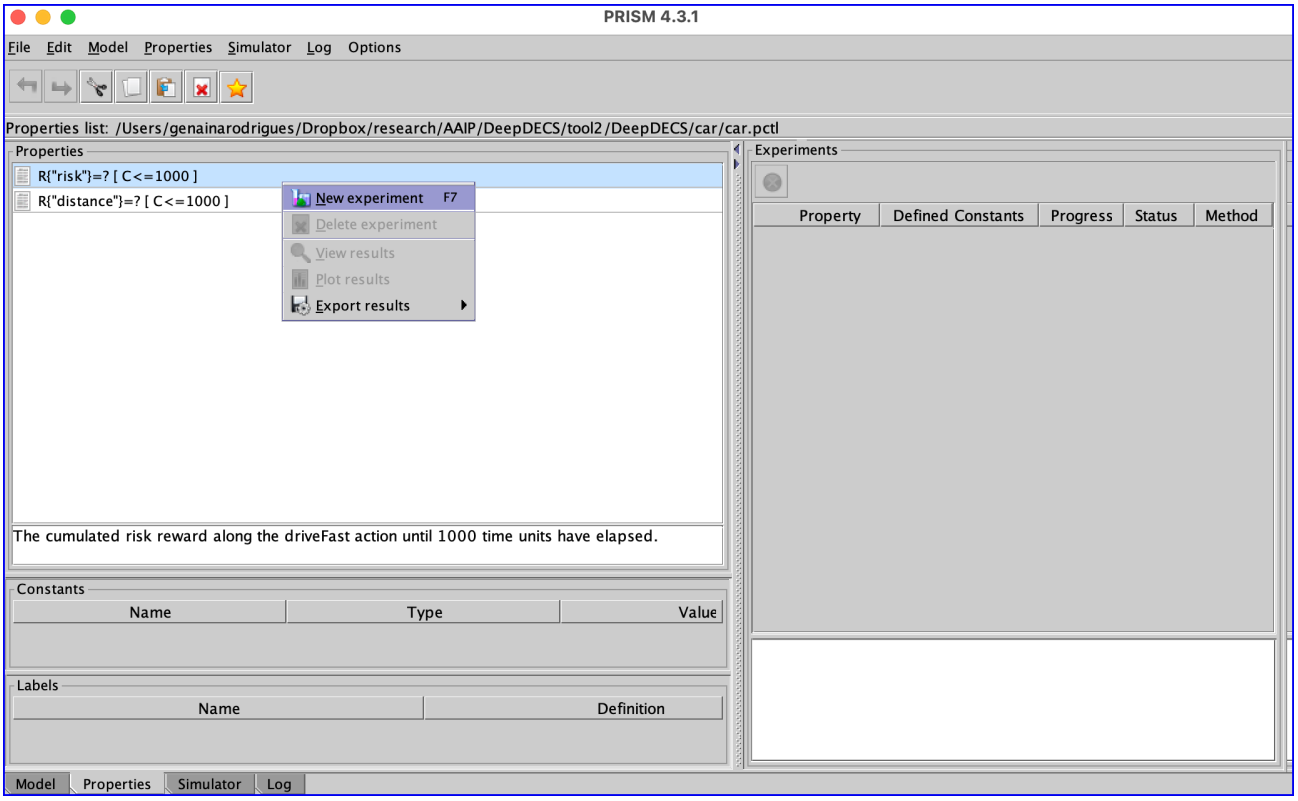


Figure 4: With the DNN-perception DTMC from Listing 3 and the PCTL properties from (1) loaded into PRISM, right-click on the "risk" property and select 'New experiment' from the pop-up menu.

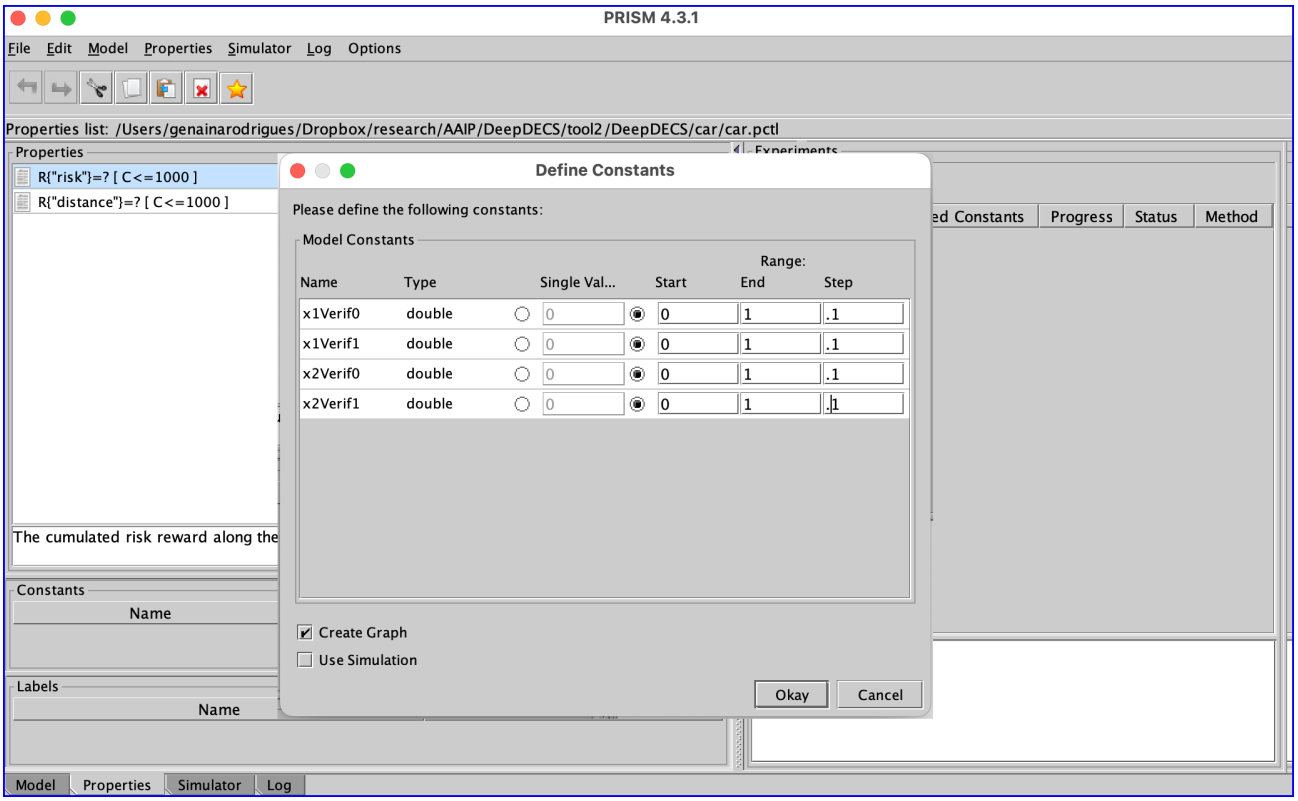


Figure 5: Select suitable ranges for the probabilities with which the controller takes each of its actions.

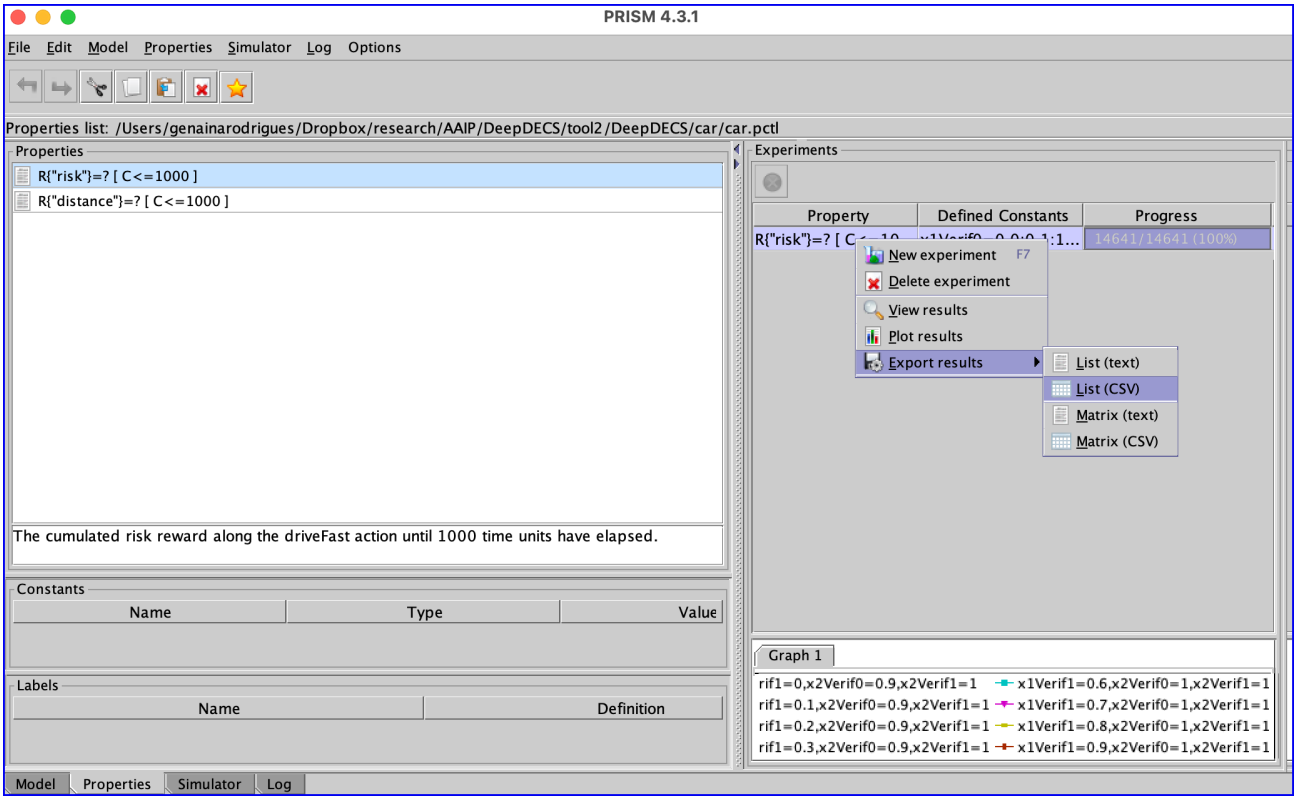
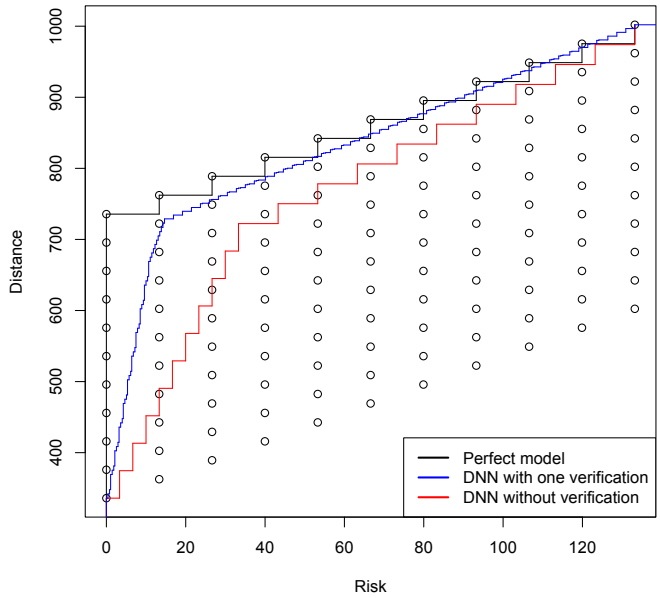


Figure 6: Export the PRISM experiment results into a .csv file.

Risk-Perf	Dist-Perf	Risk-NoVer	Dist-NoVer	Risk-Ver	Dist-Ver
0	336	0	336	0	336
13.32	362.64	9.99	363.972	9.5904	357.978
26.64	389.28	19.98	391.944	19.1808	379.956
39.96	415.92	29.97	419.916	28.7712	401.934
53.28	442.56	39.96	447.888	38.3616	423.912
66.6	469.2	49.95	475.86	47.952	445.89
79.92	495.84	59.94	503.832	57.5424	467.868
93.24	522.48	69.93	531.804	67.1328	489.846
106.56	549.12	79.92	559.776	76.7232	511.824
119.88	575.76	89.91	587.748	86.3136	533.802
133.2	602.4	99.9	615.72	95.904	555.78
0	375.96	3.33	374.628	0.3996	341.994
13.32	402.6	13.32	402.6	9.99	363.972
26.64	429.24	23.31	430.572	19.5804	385.95

(a)



(b)

Figure 7: Pareto-optimal controller synthesis for the autonomous car: (a) excerpt from the csv file with the risk and distance reward values for the three DTMC models—the perfect model, DNN without verification and the DNN with verification; (b) Pareto fronts for the autonomous car with perfect-perception, DNN-perception with no DNN verification technique, and DNN-perception with one verification technique.

References

[1] Radu Calinescu, Calum Imrie, Ravi Mangal, Corina Păsăreanu, Genaína Nunes Rodrigues, Misael Alpizar Santana, and Gricel Vázquez. Discrete-event controller synthesis for autonomous systems with deep-

```
data<-read.csv(file.choose()) #choose the file results/AutonomousCar.csv
head(data) #check if data was properly loaded
install.packages("rPref") #install the package to plot the Pareto fronts
library(rPref)  #load the package library

x<-data[,1]   #instantiate the variables of the perfect model reward values
length(x)
length(y)
y<-data[,2]
d1P<-as.data.frame(x)
d1P<-cbind(d1P,y)
head(d1P) #checking if the perfect model data was instantiated

xNV<-data[,3]  #instantiate the variables of the DNN-without-verif model reward values
length(xNV)
yNV<-data[,4]
d1NV<-as.data.frame(xNV)
d1NV<-cbind(d1NV,yNV)
head(d1NV) #checking if the DNN-without-verif data was instantiated

xWV<-data[,5] #instantiate the variables of the DNN-with-verif model reward values
length(xWV)
yWV<-data[,6]
d1WV<-as.data.frame(xWV)
d1WV<-cbind(d1WV,yWV)
head(d1WV) #checking if the DNN-with-verif data was instantiated

# plots Pareto front for the perfect model
show_front <- function(pref) {
  plot(d1P$x, d1P$y, xlab="Risk", ylab="Distance")
  sky <- psel(d1P, pref)
  plot_front(d1P, pref, col = "black")
}
show_front(low(x)  * high(y))

# plots Pareto front for the DNN-without-verif model
show_front2 <- function(pref2) {
  sky <- psel(d1NV, pref2)
  plot_front(d1NV, pref2, col = "red")
}
show_front2(low(xNV)  * high(yNV))

# plots Pareto front for the DNN-with-verif model
show_front3 <- function(pref3) {
  sky <- psel(d1WV, pref3)
  plot_front(d1WV, pref3, col = "blue")
}
show_front3(low(xWV)  * high(yWV))

legend(x = "bottomright",          # Position
       legend = c("Perfect model", "DNN without verification",
                  "DNN with one verification"),     # Legend texts
       col = c("black", "red", "blue"),           # Line colors
       lwd = 2)
```

Listing 4: R script for generating the Pareto fronts for the two optimization objectives for the autonomous car (minimize the expected risk and maximize the expected driven distance over 1000 DTMC state transitions).

learning perception components. *arXiv preprint arXiv:2202.03360*, 2023.

[2] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of the 23rd Int. Conf. on Computer Aided Verification*, volume 6806 of *LNCS*, pages

585–591. Springer, 2011.

[3] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2021.

[4] Simos Gerasimou, Radu Calinescu, and Giordano Tamburrelli. Synthesis of probabilistic models for quality-of-service software engineering. *Automated Software Engineering*, 25(4):785–831, 2018.