

January 2013

Technical Report

Probabilistic Models and Model Checking

Context Model Library for C2M2L-1

Component, Context, and Manufacturing Model Library

Prepared for BAE Systems

C2M2L

Subcontract 338724

Technical Point of Contact:

Dr. Darren Cofer

Rockwell Collins, Inc.
7805 Telegraph Rd. #100
Bloomington, MN 55438
Telephone: (319) 263-2571
ddcofer@rockwellcollins.com

Business Point of Contact:

Ms. Kelly M. Scott

Rockwell Collins, Inc.
400 Collins Rd. NE, MS 121-200
Cedar Rapids, IA 52498
Telephone: (319) 295-7323
kmscott@rockwellcollins.com

Distribution Statement C: Distribution Authorized to U.S Government Agencies and their contractors.
Other requests for this document shall be referred to the DARPA Technical Information Office.



Rockwell Collins, Inc.
400 Collins Rd. NE
Cedar Rapids, Iowa 52498

Table of Contents

1	Executive Summary.....	2
2	Drive Train Model.....	3
2.1	Drive Train Model Description.....	3
2.1.1	Throttle Driver Module.....	4
2.1.2	Diesel Engine Module.....	4
2.1.3	Transmission Module.....	5
2.1.4	Transmission Controller Module	6
2.1.5	Vehicle State Module.....	8
2.1.6	Terrain Module	8
2.1.7	Vehicle Load and Mileage Computations.....	11
2.2	Drive Train Model Properties	14
2.2.1	Vehicle Properties for Moderate Terrain	14
2.2.2	Vehicle Properties for Moderately Smooth Terrain.....	15
2.3	Drive Train Model Simulations	16
2.3.1	Vehicle Start.....	16
2.3.2	Response to Slope Increase (Level to Uphill).....	17
2.3.3	Response to Slope Decrease (Uphill to Downhill)	18
2.3.4	Response to Drastic Slope Increase (Downhill to Steep Uphill)	18
3	PRISM Terrain Modeler.....	19
3.1	The Terrain Model	19
3.1.1	Kuchar.....	19
3.1.2	Max Entropy	19
3.2	Terrain Specifications	20
3.3	Running the Tool	23
4	Assume/Guarantee.....	26
5	References	38

1 Executive Summary

Rockwell Collins supported BAE performance on DARPA C2M2L Technical Area 2 (Context Models). This report describes our work on probabilistic context models and associated probabilistic analysis methods. Our work covers three main topics:

1. Developing a probabilistic ground vehicle drive train model to interact with terrain context models
2. Creation of probabilistic terrain context models
3. Developing analysis tools for assume/guarantee-style contracts in probabilistic models.

The first topic, enhancing a ground vehicle drive train model, is described in Section 2 of this report. Our starting point was a drive train model from Vanderbilt University which was subsequently translated by Smart Information Flow Technologies (SIFT) for analysis by the PRISM tool [8]. We have made many extensions and improvements to this model to be able to tie it to context models (such as terrain) and to prove properties such as the expected mileage of the vehicle.

The second topic, creation of terrain context models, is described in Section 3 of this report. The terrain modeler is a tool that automates the generation of probabilistic terrain models for PRISM from terrain specifications provided by the user. The tool supports two different terrain models, the model derived by Kuchar [1] and the Max-Entropy model. Kuchar describes a Markov chain model for expressing probabilities of given changes in elevation. The Max-Entropy model uses the probability distribution that has maximum entropy among those permitted by the observed information. Terrain model specifications take the form of specialized PRISM comments that can be embedded in valid PRISM models. Running the terrain modeling tool on a PRISM model containing a terrain specification will result in a file containing the original PRISM model updated with an instance of the specified terrain model.

The third topic, compositional analysis of probabilistic systems using assume/guarantee reasoning, is discussed in Section 4. Compositional reasoning is a structured approach to verification that enables the analysis of systems whose size would otherwise overwhelm even modern analysis tools [5]. Compositional reasoning enables the analysis of large systems of components by first decomposing the overall problem into several smaller and more tractable problems, solving those problems, and then combining the individual results to formulate a solution for the original problem. While compositional reasoning techniques for formal verification are common and have been studied extensively, they are relatively new in the area of probabilistic reasoning. The most promising research in this area has focused on the use of adversaries to ensure probabilistic bounds on system behaviors. Even this research, however, appears to have been focused primarily on games: systems with some terminal state. We study the use of adversaries in reactive (steady state) systems and a possible approach for verifying assume/guarantee contracts for reactive systems expressed as PRISM modules.

2 Drive Train Model

2.1 Drive Train Model Description

Our starting point was a drive train model provided by Vanderbilt University which was subsequently translated by Smart Information Flow Technologies (SIFT) for analysis by the PRISM tool. We have made many extensions and improvements to this model to be able to tie it to context models (such as terrain) and to prove properties such as the expected mileage of the vehicle. To achieve this goal, we added state to keep track of the vehicle's position within a terrain model. We also added several formulas to more accurately model the load on the vehicle and to compute the instantaneous mileage. These and other aspects are described in the subsections that follow. We have not attempted to make this vehicle model 100% realistic but just close enough to provide a convincing demonstration of property verification that makes use of probabilistic context models. Users with more domain expertise can tune the parameters. Figure 1 depicts the drive train model. This depiction was created with Simulink. It helps visualize the model and keep track of the names of the inputs and outputs to the PRISM modules¹.

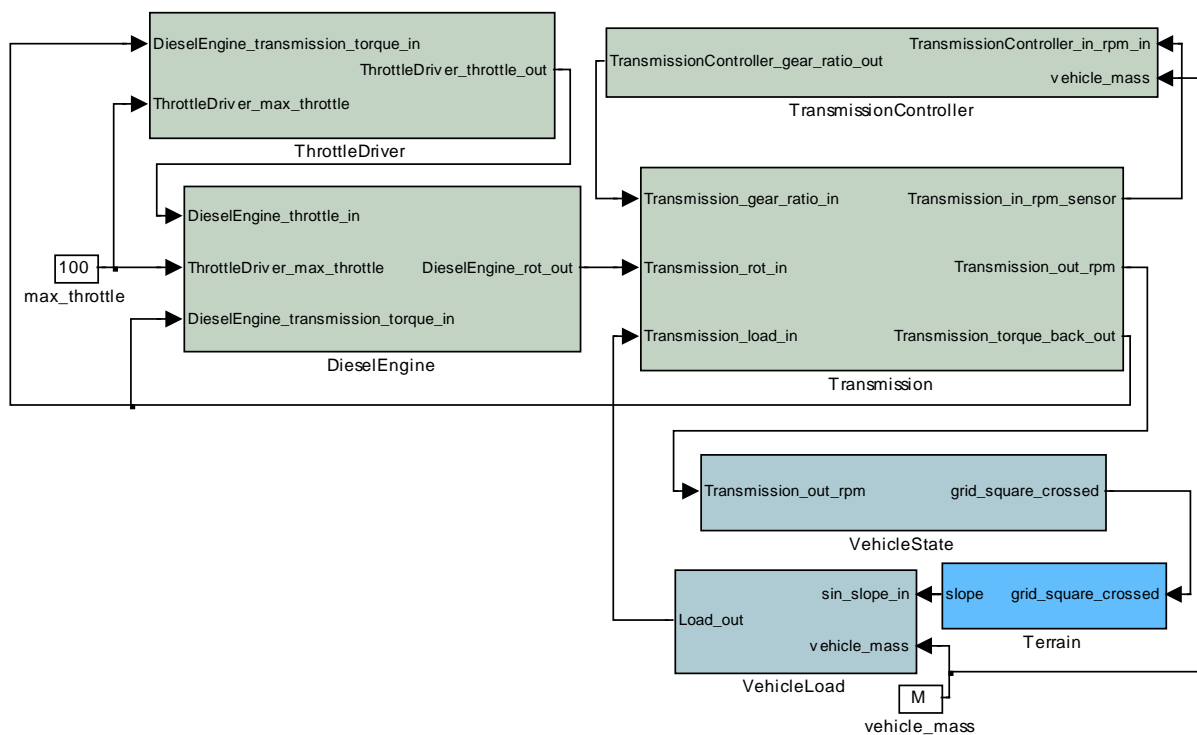


Figure 1 – Drive Train Model Depicted in Simulink®

¹ VehicleLoad is not technically a PRISM module since it does not contain any state variables. It is a set of formulas. The other modules have at least some state (even if it is just to keep track of failure modes).

2.1.1 Throttle Driver Module

The Throttle Driver Module computes the throttle value (ThrottleDriver_throttle_out), which is used by the Diesel Engine Module to compute the engine rpm (DieselEngine_rot_out). The PRISM code is shown in Figure 4. We have experimented with fixed and variable throttles. This version of the vehicle model has a variable throttle with range 0-100%, but only increments of 5% are used (5%, 10%, and so on).

```
const int ThrottleDriver_max_throttle = 100;

formula throttle_goal = max(min(ThrottleDriver_max_throttle,
(DieselEngine_transmission_torque_in + 1500) / DieselEngine_throttle_scale),0);

module ThrottleDriver
    throttle: [0..ThrottleDriver_max_throttle] init 15;
    [tic] (throttle <= throttle_goal - 5) -> (throttle' = throttle + 5);
    [tic] (throttle >= throttle_goal + 5) -> (throttle' = throttle - 5);
    [tic] (throttle < throttle_goal + 5 & throttle > throttle_goal - 5) -> true;
endmodule

formula ThrottleDriver_throttle_out = throttle;
```

Figure 2 – PRISM Code for Throttle Driver Module

2.1.2 Diesel Engine Module

This module computes the engine output in rpm (DieselEngine_rot_out) as the product of the throttle scale (DieselEngine_throttle_scale, equal to 120) and throttle input (DieselEngine_throttle_in) minus the torque back to the engine (DieselEngine_transmission_torque_in). A simplified version (without modes) of the Diesel Engine Module is depicted in Figure 3. The PRISM code is shown in Figure 4.

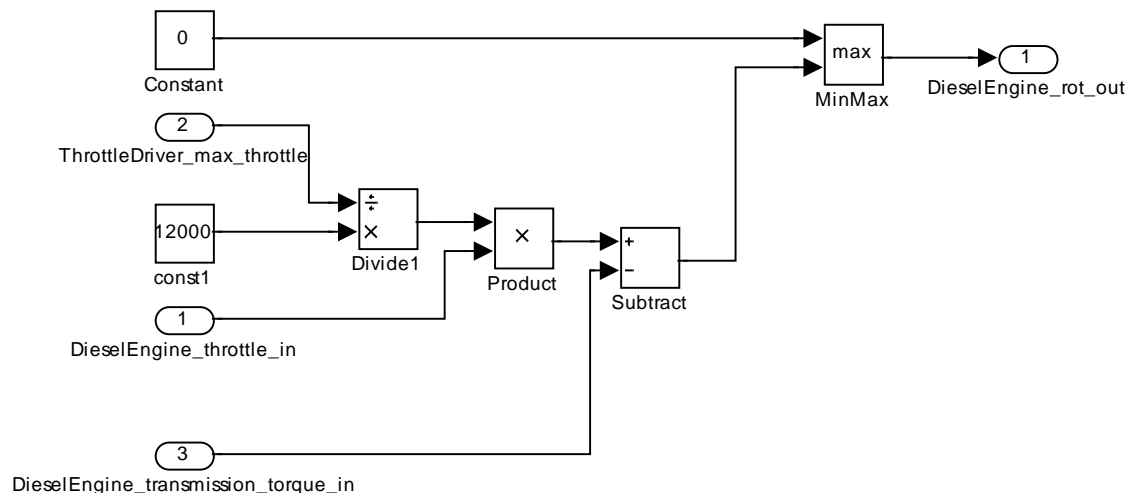


Figure 3 – Simulink® Depiction of Simplified Diesel Engine Module

```

const int DieselEngine_mode_nominal = 0;
const int DieselEngine_mode_failed = 1;
const double DieselEngine_fail_rate = 0;

const double DieselEngine_throttle_scale = 12000.0 / ThrottleDriver_max_throttle;

formula DieselEngine_rot_out =
  (DieselEngine_mode != DieselEngine_mode_nominal) ? 0 :
    max(0, DieselEngine_throttle_scale * DieselEngine_throttle_in -
      DieselEngine_transmission_torque_in);

module DieselEngine
  DieselEngine_mode: [0..1] init DieselEngine_mode_nominal;
  [tic] DieselEngine_mode = DieselEngine_mode_nominal ->
    DieselEngine_fail_rate : (DieselEngine_mode' = DieselEngine_mode_failed)
    + 1-DieselEngine_fail_rate: true;
endmodule

```

Figure 4 – PRISM Code for Diesel Engine Module

2.1.3 Transmission Module

The Transmission Module computes the transmission output in rpm (Transmission_out_rpm) as the product of the engine rpm (Transmission_rot_in) and the gear ratio (Transmission_gear_ratio_in) provided by the Transmission Controller Module. It also computes the torque back to the engine (Transmission_torque_back_out) as the product of the gear ratio (Transmission_gear_ratio_in) and the load on the vehicle (Transmission_load_in). A simplified version (without modes) of the Transmission Module is depicted in Figure 5. The PRISM code is shown in Figure 6.

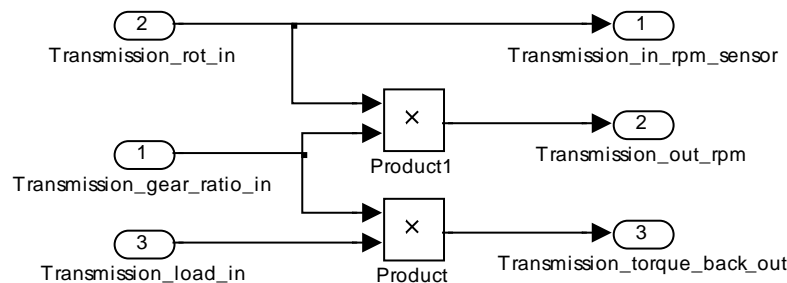


Figure 5 – Simulink® Depiction of Simplified Transmission Module

```

const int Transmission_mode_nominal = 0;
const int Transmission_mode_sensor_fail = 1;
const double Transmission_out_rpm_speed_limit = 2000000;
const int Transmission_mode_out_rpm_over_limit = 2;
const double Transmission_sensor_fail_rate = 0.0001;

formula Transmission_out_rpm =
    (Transmission_mode != Transmission_mode_nominal) ? 0 :
    Transmission_rot_in * Transmission_gear_ratio_in;

formula Transmission_out_rpm_sensor =
    (Transmission_mode != Transmission_mode_nominal) ? 0 : Transmission_out_rpm;

formula Transmission_in_rpm_sensor =
    (Transmission_in_sensor_mode != Transmission_mode_nominal) ? 0 :
    Transmission_rot_in;

formula Transmission_torque_back_out = Transmission_gear_ratio_in *
    Transmission_load_in;

module Transmission
    Transmission_mode: [0..2] init Transmission_mode_nominal;
    Transmission_in_sensor_mode: [0..1] init Transmission_mode_nominal;
    Transmission_out_sensor_mode: [0..1] init Transmission_mode_nominal;
    [tic] Transmission_out_sensor_mode != Transmission_mode_nominal -> true;
    [tic] Transmission_out_sensor_mode = Transmission_mode_nominal &
        Transmission_out_rpm <= Transmission_out_rpm_speed_limit ->
        Transmission_sensor_fail_rate : (Transmission_out_sensor_mode' =
        Transmission_mode_sensor_fail) + 1-Transmission_sensor_fail_rate: true;
    [tic] Transmission_out_sensor_mode = Transmission_mode_nominal &
        Transmission_out_rpm > Transmission_out_rpm_speed_limit ->
        1: (Transmission_mode' = Transmission_mode_out_rpm_over_limit);
endmodule

```

Figure 6 – PRISM Code for Transmission Module

2.1.4 Transmission Controller Module

The Transmission Controller Module computes the gear ratio (TransmissionController_gear_ratio_out), which is used by the Transmission Module. There are six gears (and corresponding gear ratios) defined in the vehicle model. Lower and upper bounds for gear shifting are defined (G1Low, G1High, etc.). The transmission upshifts when the transmission rpm is higher than the upper bound for the current gear. The transmission downshifts when the transmission rpm is lower than the lower bound for the current gear. The Transmission Controller Module includes a counter ('ctr') that restricts the gear to shifting every other step/tic. This prevents the throttle and gear from "co-chatter." The PRISM code for the Transmission Controller Module is shown in Figure 7.

```

formula level_grade_load = P_inertia + P_tires + P_accessories;

const double G1 = 0.235849057;
const double G2 = 0.327868852;
const double G3 = 0.431034483;
const double G4 = 0.598802395;
const double G5 = 1;
const double G6 = 1.388888889;

const double ShiftRPM = 1800;
const double G1Low = 0;
const double G2Low = ShiftRPM * G1;
const double G1High = ShiftRPM * (G1 + G2)/2 + level_grade_load*(G2-G1);
const double G3Low = ShiftRPM * G2;
const double G2High = ShiftRPM * (G2 + G3)/2 + level_grade_load*(G3-G2);
const double G4Low = ShiftRPM * G3;
const double G3High = ShiftRPM * (G3 + G4)/2 + level_grade_load*(G4-G3);
const double G5Low = ShiftRPM * G4;
const double G4High = ShiftRPM * (G4 + G5)/2 + level_grade_load*(G5-G4);
const double G6Low = ShiftRPM * G5;
const double G5High = ShiftRPM * (G5 + G6)/2 + level_grade_load*(G6-G5);
const double G6High = ShiftRPM * G6;

const int TransmissionController_mode_nominal = 0;
const int TransmissionController_mode_failed = 1;
//const double TransmissionController_fail_rate = 0.0001;
const double TransmissionController_fail_rate = 0;

formula TransmissionController_gear_ratio_out =
    TransmissionController_mode != TransmissionController_mode_nominal ? G1 :
        TransmissionController_gear = 1 ? G1 :
        TransmissionController_gear = 2 ? G2 :
        TransmissionController_gear = 3 ? G3 :
        TransmissionController_gear = 4 ? G4 :
        TransmissionController_gear = 5 ? G5 :
        TransmissionController_gear = 6 ? G6 : 1;

module TransmissionController
    TransmissionController_mode: [0..1] init TransmissionController_mode_nominal;
    TransmissionController_gear: [1..6] init 1;
    ctr: [0..1] init 0;
    [tic] (ctr = 1) & (TransmissionController_mode = TransmissionController_mode_nominal)
    ->
        TransmissionController_fail_rate :
            (ctr' = 0) & (TransmissionController_mode' = TransmissionController_mode_failed) +
            1-TransmissionController_fail_rate :
                (ctr' = 0) & (TransmissionController_gear' =
                    TransmissionController_mode != TransmissionController_mode_nominal ? 1 :
                    (TransmissionController_gear = 1)&(TransmissionController_in_rpm_in > G1High) ? 2:
                    (TransmissionController_gear = 2)&(TransmissionController_in_rpm_in < G2Low ) ? 1:
                    (TransmissionController_gear = 2)&(TransmissionController_in_rpm_in > G2High) ? 3:
                    (TransmissionController_gear = 3)&(TransmissionController_in_rpm_in < G3Low ) ? 2:
                    (TransmissionController_gear = 3)&(TransmissionController_in_rpm_in > G3High) ? 4:
                    (TransmissionController_gear = 4)&(TransmissionController_in_rpm_in < G4Low ) ? 3:
                    (TransmissionController_gear = 4)&(TransmissionController_in_rpm_in > G4High) ? 5:
                    (TransmissionController_gear = 5)&(TransmissionController_in_rpm_in < G5Low ) ? 4:
                    (TransmissionController_gear = 5)&(TransmissionController_in_rpm_in > G5High) ? 6:
                    (TransmissionController_gear = 6)&(TransmissionController_in_rpm_in < G6Low ) ? 5:
                    TransmissionController_gear);
    [tic] (ctr != 1) -> (ctr' = 1);
endmodule

```

Figure 7 – PRISM Code for Transmission Controller Module

2.1.5 Vehicle State Module

The resolution of the terrain models we are using is 100 m. In other words, we have one elevation value for every 100 m x 100 m square in the grid. Since the elevation state (and slope) should only update when a grid square boundary is crossed, we added a Vehicle State Module to keep track of the vehicle's approximate position within the current 100 m x 100 m square of the Terrain Module. The Vehicle State Module sets 'grid_square_crossed' to 1 whenever a grid square boundary is crossed. The elevation state variable (theNextBin) in the terrain module updates if and only if 'grid_square_crossed' is 1. Also, 'delta_position' is used to compute the velocity of the vehicle, which is used in the mileage computation. The PRISM code for the Vehicle State Module is shown in Figure 8.

```
const double Wheel_diameter = 1; //in meters
const double pi = 3.14;
const double time_step = 1; //in seconds
const int grid_size = 100; //in meters

formula Differential_out_rpm = Transmission_out_rpm/10; //Differential
formula delta_position =
floor(Differential_out_rpm*Wheel_diameter*pi*time_step/60);
formula num_crossings = floor((position + delta_position)/grid_size);
formula grid_square_crossed = (num_crossings > 0);

module VehicleState
    position : [0..grid_size] init 0;
    [tic] true -> (position' = position + delta_position -
num_crossings*grid_size);
endmodule
```

Figure 8 – PRISM Code for Vehicle State Module

2.1.6 Terrain Module

The PRISM code for the Terrain Module corresponding to the Moderate Terrain Category is shown in Figure 9. This Terrain Module is defined based on J. K. Kuchar's paper [1] and was created using the PRISM Terrain Modeler we developed (see Section 3). To keep our model manageable, we chose to consider 15 altitude bins in each category. At the upper and lower ends of the altitude range, the transition probabilities are chosen to ensure that the probabilities in each row of the transition matrix sum to 1. Several stand-alone terrain models were delivered in November (some as PRISM files, others as Excel workbooks).

```
const int step = 100;

module terrainModule
    // Sigma          342          Terrain Model Parameter
    // Beta           1.3e-3       Terrain Model Parameter
    // maxElevation    1400        Upper bound on elevation
    // minElevation    1100        Lower bound on elevation
    // binCount        15          Number of discrete elevations
    // tickName        tic         Process synchronization
    // currBinName      theNowBin    STATE: slopeModule state
    // nextBinName      theNextBin  STATE: terrainModule state
    // currElevationName currentElevation OUTPUT: Name of the current elevation signal
```

```

// nextElevationName      nextElevation      OUTPUT: Name of the next elevation signal
// currSlope              slope              OUTPUT: Name of the slope signal
// updateElevationName    grid_square_crossed INPUT: Name of the "step" signal
theNextBin: [0..14];
[tic] ((grid_square_crossed) &
      (nextElevation <= 1120.0)) ->
      0.2201: (theNextBin' = 1) + // 1130.0
      0.0340: (theNextBin' = 2) + // 1150.0
      0.0016: (theNextBin' = 3) + // 1170.0
      0.7443: (theNextBin' = 0); // 1110.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1120.0) &
      (nextElevation <= 1140.0)) ->
      0.3122: (theNextBin' = 0) + // 1110.0
      0.2198: (theNextBin' = 2) + // 1150.0
      0.0339: (theNextBin' = 3) + // 1170.0
      0.0016: (theNextBin' = 4) + // 1190.0
      0.4325: (theNextBin' = 1); // 1130.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1140.0) &
      (nextElevation <= 1160.0)) ->
      0.0510: (theNextBin' = 0) + // 1110.0
      0.2617: (theNextBin' = 1) + // 1130.0
      0.2194: (theNextBin' = 3) + // 1170.0
      0.0338: (theNextBin' = 4) + // 1190.0
      0.0016: (theNextBin' = 5) + // 1210.0
      0.4325: (theNextBin' = 2); // 1150.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1160.0) &
      (nextElevation <= 1180.0)) ->
      0.0027: (theNextBin' = 0) + // 1110.0
      0.0484: (theNextBin' = 1) + // 1130.0
      0.2622: (theNextBin' = 2) + // 1150.0
      0.2190: (theNextBin' = 4) + // 1190.0
      0.0337: (theNextBin' = 5) + // 1210.0
      0.0016: (theNextBin' = 6) + // 1230.0
      0.4324: (theNextBin' = 3); // 1170.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1180.0) &
      (nextElevation <= 1200.0)) ->
      0.0027: (theNextBin' = 1) + // 1130.0
      0.0486: (theNextBin' = 2) + // 1150.0
      0.2625: (theNextBin' = 3) + // 1170.0
      0.2187: (theNextBin' = 5) + // 1210.0
      0.0336: (theNextBin' = 6) + // 1230.0
      0.0015: (theNextBin' = 7) + // 1250.0
      0.4324: (theNextBin' = 4); // 1190.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1200.0) &
      (nextElevation <= 1220.0)) ->
      0.0027: (theNextBin' = 2) + // 1150.0
      0.0487: (theNextBin' = 3) + // 1170.0
      0.2629: (theNextBin' = 4) + // 1190.0
      0.2183: (theNextBin' = 6) + // 1230.0
      0.0335: (theNextBin' = 7) + // 1250.0
      0.0015: (theNextBin' = 8) + // 1270.0
      0.4324: (theNextBin' = 5); // 1210.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1220.0) &
      (nextElevation <= 1240.0)) ->
      0.0027: (theNextBin' = 3) + // 1170.0
      0.0489: (theNextBin' = 4) + // 1190.0
      0.2633: (theNextBin' = 5) + // 1210.0
      0.2180: (theNextBin' = 7) + // 1250.0
      0.0334: (theNextBin' = 8) + // 1270.0
      0.0015: (theNextBin' = 9) + // 1290.0
      0.4322: (theNextBin' = 6); // 1230.0
[tic] ((grid_square_crossed) &
      (nextElevation > 1240.0) &
      (nextElevation <= 1260.0)) ->
      0.0028: (theNextBin' = 4) + // 1190.0

```

```

0.0489: (theNextBin' = 5) + // 1210.0
0.2637: (theNextBin' = 6) + // 1230.0
0.2176: (theNextBin' = 8) + // 1270.0
0.0333: (theNextBin' = 9) + // 1290.0
0.0015: (theNextBin' = 10) + // 1310.0
0.4322: (theNextBin' = 7); // 1250.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1260.0) &
(nextElevation <= 1280.0)) ->
0.0028: (theNextBin' = 5) + // 1210.0
0.0491: (theNextBin' = 6) + // 1230.0
0.2640: (theNextBin' = 7) + // 1250.0
0.2172: (theNextBin' = 9) + // 1290.0
0.0332: (theNextBin' = 10) + // 1310.0
0.0015: (theNextBin' = 11) + // 1330.0
0.4322: (theNextBin' = 8); // 1270.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1280.0) &
(nextElevation <= 1300.0)) ->
0.0028: (theNextBin' = 6) + // 1230.0
0.0493: (theNextBin' = 7) + // 1250.0
0.2644: (theNextBin' = 8) + // 1270.0
0.2168: (theNextBin' = 10) + // 1310.0
0.0331: (theNextBin' = 11) + // 1330.0
0.0015: (theNextBin' = 12) + // 1350.0
0.4321: (theNextBin' = 9); // 1290.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1300.0) &
(nextElevation <= 1320.0)) ->
0.0028: (theNextBin' = 7) + // 1250.0
0.0494: (theNextBin' = 8) + // 1270.0
0.2648: (theNextBin' = 9) + // 1290.0
0.2166: (theNextBin' = 11) + // 1330.0
0.0329: (theNextBin' = 12) + // 1350.0
0.0015: (theNextBin' = 13) + // 1370.0
0.4320: (theNextBin' = 10); // 1310.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1320.0) &
(nextElevation <= 1340.0)) ->
0.0028: (theNextBin' = 8) + // 1270.0
0.0496: (theNextBin' = 9) + // 1290.0
0.2651: (theNextBin' = 10) + // 1310.0
0.2162: (theNextBin' = 12) + // 1350.0
0.0328: (theNextBin' = 13) + // 1370.0
0.0015: (theNextBin' = 14) + // 1390.0
0.4320: (theNextBin' = 11); // 1330.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1340.0) &
(nextElevation <= 1360.0)) ->
0.0028: (theNextBin' = 9) + // 1290.0
0.0497: (theNextBin' = 10) + // 1310.0
0.2655: (theNextBin' = 11) + // 1330.0
0.2158: (theNextBin' = 13) + // 1370.0
0.0342: (theNextBin' = 14) + // 1390.0
0.4320: (theNextBin' = 12); // 1350.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1360.0) &
(nextElevation <= 1380.0)) ->
0.0028: (theNextBin' = 10) + // 1310.0
0.0499: (theNextBin' = 11) + // 1330.0
0.2659: (theNextBin' = 12) + // 1350.0
0.2495: (theNextBin' = 14) + // 1390.0
0.4319: (theNextBin' = 13); // 1370.0
[ tic] ((grid_square_crossed) &
(nextElevation > 1380.0)) ->
0.0028: (theNextBin' = 11) + // 1330.0
0.0501: (theNextBin' = 12) + // 1350.0
0.2662: (theNextBin' = 13) + // 1370.0
0.6809: (theNextBin' = 14); // 1390.0
[ tic] (! grid_square_crossed) -> true;
endmodule

```

```

module slopeModule
  theNowBin: [0..14];
  [tic] ( grid_square_crossed) -> (theNowBin' = theNextBin);
  [tic] (! grid_square_crossed) -> true;
endmodule

/// Generated Terrain Formula
formula currentElevation = 1110.0 + 20.0*theNowBin;
/// Generated Terrain Formula
formula nextElevation = 1110.0 + 20.0*theNextBin;
/// Generated Terrain Formula
formula slope = ((nextElevation - currentElevation)/step);

// We really want sin(slope_angle) for the P_grade formula.
// Note: sin(slope_angle)=rise/(sqrt(rise^2+run^2)).
// Since PRISM can't do a square root, we use a table for possible slope values (not all
used).

formula sin_slope =
  slope = -1 ? -.7071 :
  slope = -.8 ? -.6247 :
  slope = -.6 ? -.5145 :
  slope = -.4 ? -.3714 :
  slope = -.2 ? -.1961 :
  slope = 0 ? 0 :
  slope = .2 ? .1961 :
  slope = .4 ? .3714 :
  slope = .6 ? .5145 :
  slope = .8 ? .6247 :
  slope = 1 ? .7071 : 0;

// Divide by a factor of 4 (assume vehicle will take a longer, smoother path as needed).
// This prevents the vehicle from excessive stalling.
formula sin_slope_in = sin_slope/4;

```

Figure 9 – PRISM Code for (Moderate) Terrain Module (spans multiple pages)

2.1.7 Vehicle Load and Mileage Computations

This portion of the PRISM model computes the instantaneous load on the vehicle as well as the instantaneous mileage and range. The mileage and vehicle range rewards can be used to obtain the expected vehicle mileage and range over the terrain. The primary sources for the formulas and constants used in this module are [2] and [3]. The instantaneous load on the vehicle is

$$P_{\text{inertia}} + P_{\text{grade}} + P_{\text{air_drag}} + P_{\text{tires}} + P_{\text{accessories}},$$

where

- $P_{\text{inertia}} = 0.5 * 1.03 * \text{vehicle_mass} * \text{acceleration}$ is the power to overcome inertia,
- $P_{\text{grade}} = \text{vehicle_mass} * \text{gravity} * \sin(\text{slope angle})$ is the power to overcome the load from grade,
- $P_{\text{air_drag}}$ is the power to overcome air drag,
- P_{tires} is the power to overcome rolling resistance, and
- $P_{\text{accessories}}$ is the power used by the vehicle accessories (such as lights, radio, air conditioning, etc.).

In this first-order vehicle model, we assume the vehicle acceleration is 1 (constant velocity) and that $P_{\text{accessories}}$ is 0.

If the vehicle is moving (nonzero velocity), then the vehicle mileage can be estimated by

$$(\text{energy_per_gallon} * \text{engine_efficiency}) / (\text{instantaneous load}),$$

where energy_per_gallon is the amount of energy in a gallon of diesel (150,150,000 joules) and engine_efficiency is the fraction of the fuel energy converted to work by the engine and delivered to the transmission. We use 18% for the engine efficiency in this model.

Note that no state variables are directly used in the used in the computation of vehicle load or mileage. However, these do rely on velocity, which is computed from the state variable 'position' of the Vehicle State Module.

```

const double vehicle_mass = 23991; // kilograms
const double gravity = 9.81;

// Instantaneous load is P_inertia + P_grade + P_air_drag + P_tires + P_accessories
formula P_inertia = 0.5 * 1.03 * vehicle_mass * 1; // Assume constant velocity
formula P_grade = vehicle_mass * gravity * sin_slope_in;

const double drag_coefficient = .6; // Drag coefficient for SUVs is 50-60%.
const double air_density = 1.3; // At sea level, the air density is 1.3 kg/m^3
const double vehicle_frontal_area = 10.8; // Assumes 3.6 meters wide and 3 meters tall
const double rolling_friction_coefficient = 0.01;

formula P_air_drag =
    .5*drag_coefficient*air_density*vehicle_frontal_area*pow(velocity,2);

formula P_tires = rolling_friction_coefficient*vehicle_mass*gravity;

formula P_accessories = 0; //Assume this is negligible

formula Load_out = max(0,P_inertia + P_grade + P_tires + P_accessories);

formula Transmission_load_in = Load_out;

// Mileage Computation
const double energy_per_gallon = 150150000; // joules in a gallon of diesel
const double fuel_capacity = 171; // gallons (used to compute vehicle range)
const double engine_efficiency = .18; // used in BAE Fuel Efficiency example

// Max time a gallon of diesel can last (used for idle scenarios)
const double gallon_max_time = 2*60*60; // seconds. Assumes min burn of 1/2 gal per hr

formula velocity = delta_position/time_step; // meters/second

// If velocity is 0, gas mileage is 0. Otherwise, gas mileage is the minimum of the distance
// travelled in 2 hours and the output of the formula based on inertia, grade, drag, rolling
// resistance (tires), and accessories.
formula mileage_meters_per_gal = (velocity = 0) ? 0 : min(velocity*gallon_max_time,
    (energy_per_gallon*engine_efficiency)/(Load_out + P_air_drag));

const double meters_per_mile = 1609.344;
formula mileage_miles_per_gal = mileage_meters_per_gal/meters_per_mile;
formula velocity_miles_per_hour = velocity*60*60/meters_per_mile;
formula range_miles_per_tank = mileage_miles_per_gal*fuel_capacity;

rewards "mileage"
    true : mileage_miles_per_gal;
endrewards

rewards "vehicle_range"
    true: range_miles_per_tank;
endrewards

```

Figure 10 – PRISM Code for Vehicle Load and Mileage Computations

2.2 Drive Train Model Properties

2.2.1 Vehicle Properties for Moderate Terrain

We now prove some properties and do some probabilistic computations using PRISM on the drive train model with the Moderate Terrain Module.

2.2.1.1 Ranges of Variables

Ranges of various variables in a PRISM model can be easily computed using a PRISM property of the following form:

```
filter(range, variable, true)
```

Using properties of this form, we were able to compute the following:

- Range of velocity (velocity_miles_per_hour): 0-89 mph
- Range of engine rpm (DieselEngine_rot_out): 0-12000 rpm
- Range of slope (slope): -60% to 60%
- Range of gear (TransmissionController_gear): 1-5

Note that the upper ends for velocity and engine rpm are quite high for a diesel military ground vehicle. Recall that we have not attempted to make this vehicle model 100% realistic but just close enough to provide a convincing demonstration of property verification that makes use of probabilistic context models. We can compute the steady-state probabilities that these variables are less than some threshold. In particular, we find that the value of engine rpm is less than 4000 with a probability of 98.7% and that the vehicle's velocity is less than 5 mpg with a probability of 82.9%. The following properties in PRISM were used to make these computations.

```
S=? [ DieselEngine_rot_out<=4000 ]
S=? [ velocity_miles_per_hour<5 ]
```

2.2.1.2 Stalling

The probability of the vehicle stalling—more precisely, the percentage of all possible states for which the engine rpm (DieselEngine_rot_out) is less than 900 is 2.9%.

This was computed using the following formula and reward structure in PRISM:

```
formula engine_stall = (DieselEngine_rot_out < 900) ? 1 : 0;
rewards "stall"
    true : engine_stall;
endrewards
```

The property in PRISM is written as follows:

```
R{"stall"}=? [ S ]
```

2.2.1.3 Mileage and Range

The instantaneous mileage of the vehicle is computed using the following reward structure in PRISM:

```
rewards "mileage"
    true : mileage_miles_per_gal;
endrewards
```

The expected mileage of the vehicle is 1.4 mpg and is computed with the PRISM property written as follows:

$$R\{\text{"mileage"}\}=? \text{ [S]}$$

The instantaneous range of the vehicle is similarly computed using the following reward structure in PRISM:

```
rewards "vehicle_range"
    true: range_miles_per_tank;
endrewards
```

The expected range of the vehicle is 238 miles (assuming a 171 gallon tank). This could also be computed directly (external to PRISM) from the expected mileage as $\text{mileage} \times 171$, rather than using a separate reward structure.

2.2.2 Vehicle Properties for Moderately Smooth Terrain

Using the Moderately Smooth Terrain Module, we get slightly different results for some of the properties. The PRISM statements are not included in this section since they are the same as in the previous section (Vehicle Properties for Moderate Terrain).

2.2.2.1 Ranges of Variables

- Range of velocity (velocity_miles_per_hour): 0-54 mph
- Range of engine rpm (DieselEngine_rot_out): 0-10200 rpm
- Range of slope (slope): -40% to 40%
- Range of gear (TransmissionController_gear): 1-4

Note that the upper ends for velocity and engine rpm are quite high for a diesel military ground vehicle. We can compute the steady-state probabilities that these variables are less than some threshold. In particular, we find that the value of engine rpm is less than 4000 with a probability of 99.4% and that the vehicle's velocity is less than 5 mpg with a probability of 86.2%.

2.2.2.2 Stalling

The probability of the vehicle stalling—more precisely, the percentage of all possible states for which the engine rpm (DieselEngine_rot_out) is less than 900 is 1.3%.

2.2.2.3 Mileage and Range

The expected mileage of the vehicle is 1.2 mpg. The expected range of the vehicle is 208 miles.

2.3 Drive Train Model Simulations

In this section, we provide some screenshots from simulations of the vehicle model in PRISM (for the Moderate Terrain Category). First we show the full view of PRISM's simulation interface in Figure 11. One can choose the number of steps to simulate on one click by entering that number in the upper left text box. One can also force a transition for the next simulation step by double clicking that transition in the "Manual exploration" section. The results of all state valuables and rewards are shown in the large table at the bottom.

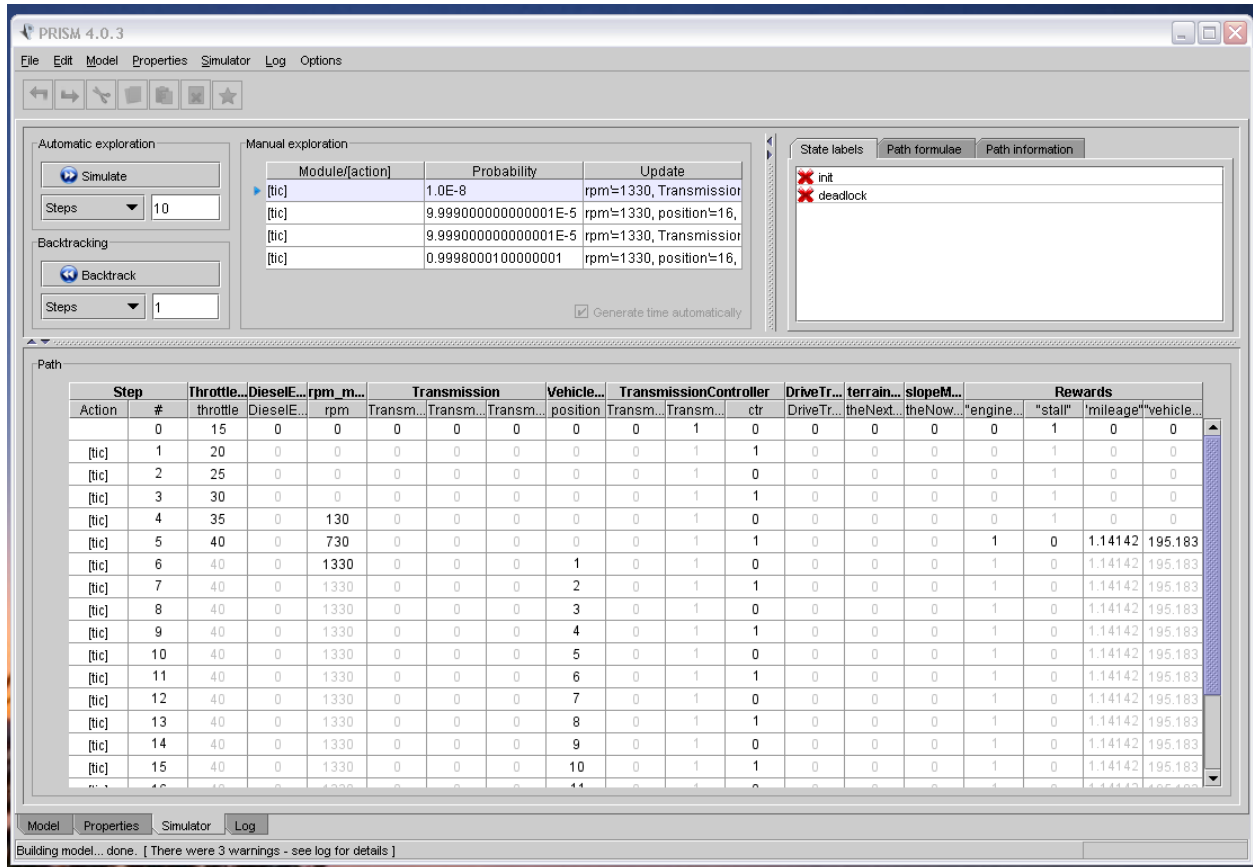


Figure 11 – PRISM Simulation Interface

2.3.1 Vehicle Start

Figure 12 is a zoomed-in version of Figure 11. This shows the first 15 steps of a simulation from the initial state. Notice how the throttle increases from 15% to 40% before rpms are sufficiently high. The rpm_monitor shows the value of engine rpm from the previous step. (There is a one-step delay since this is a state variable updated with the value of a formula.) The position does not begin to change until step 6. Since 'position' is a state variable, it must be an integer, so it is the floor of a real number. Until the floor is 1, the vehicle "doesn't move." The rewards from left to right, are "engine_sweet_spot" which is 1 whenever engine rpm is between 1100 and 1800. The second reward is "stall" which is 1 whenever the engine rpm is less than 900. The "mileage" reward is the instantaneous mileage of the vehicle in

miles per gallon. The last reward, “vehicle_range” is the instantaneous range of the vehicle in miles per tank, where the fuel capacity of the vehicle is defined to be 171 gallons.

p	#	Throttle...	DieselE...	rpm_m...	Transmission			Vehicle...	TransmissionController			DriveTr...	terrain...	slopeM...	Rewards			
		throttle	DieselE...	rpm	Transm...	Transm...	Transm...	position	Transm...	Transm...	ctr	DriveTr...	theNext...	theNow...	"engine...	"stall"	"mileage"	"vehicle..."
0	15	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
1	20	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
2	25	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
3	30	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
4	35	0	130	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
5	40	0	730	0	0	0	0	0	0	1	1	0	0	0	1	0	1.14142	195.183
6	40	0	1330	0	0	0	0	1	0	1	0	0	0	0	1	0	1.14142	195.183
7	40	0	1330	0	0	0	0	2	0	1	1	0	0	0	1	0	1.14142	195.183
8	40	0	1330	0	0	0	0	3	0	1	0	0	0	0	1	0	1.14142	195.183
9	40	0	1330	0	0	0	0	4	0	1	1	0	0	0	1	0	1.14142	195.183
10	40	0	1330	0	0	0	0	5	0	1	0	0	0	0	1	0	1.14142	195.183
11	40	0	1330	0	0	0	0	6	0	1	1	0	0	0	1	0	1.14142	195.183
12	40	0	1330	0	0	0	0	7	0	1	0	0	0	0	1	0	1.14142	195.183
13	40	0	1330	0	0	0	0	8	0	1	1	0	0	0	1	0	1.14142	195.183
14	40	0	1330	0	0	0	0	9	0	1	0	0	0	0	1	0	1.14142	195.183
15	40	0	1330	0	0	0	0	10	0	1	1	0	0	0	1	0	1.14142	195.183

Figure 12 – PRISM Simulation for Vehicle Model on Moderate Terrain (Starting out)

```

module rpm_monitor
  rpm: [0 .. 12000];
  [tic] true -> (rpm' = floor(DieselEngine_rot_out));
endmodule

```

2.3.2 Response to Slope Increase (Level to Uphill)

Figure 13 shows the vehicle model’s response to an increase in slope (from level to uphill). Note that the engine rpm drops to 0 when the slope first changes, and the vehicle stalls. Then the throttle increases until engine rpm becomes positive and the vehicle recovers from stall. Note that this is a first-order model of a vehicle and that we don’t actually anticipate the vehicle to stall in the field.

ep	#	Throttle...	DieselE...	rpm_m...	Transmission			Vehicle...	TransmissionController			DriveTr...	terrain...	slopeM...	Rewards			
		throttle	DieselE...	rpm	Transm...	Transm...	Transm...	position	Transm...	Transm...	ctr	DriveTr...	theNext...	theNow...	"engine...	"stall"	"mileage"	"vehicle..."
103	40	0	1330	0	0	0	0	98	0	1	1	0	0	0	1	0	1.14142	195.183
104	40	0	1330	0	0	0	0	99	0	1	0	0	0	0	1	0	1.14142	195.183
105	40	0	1330	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
106	45	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0
107	50	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
108	55	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0
109	60	0	409	0	0	0	0	0	0	1	1	0	1	0	0	0	0.639734	109.395
110	60	0	1009	0	0	0	0	1	0	1	0	0	1	0	0	0	0.639734	109.395
111	60	0	1009	0	0	0	0	2	0	1	1	0	1	0	0	0	0.639734	109.395
112	60	0	1009	0	0	0	0	3	0	1	0	0	1	0	0	0	0.639734	109.395
113	60	0	1009	0	0	0	0	4	0	1	1	0	1	0	0	0	0.639734	109.395
114	60	0	1009	0	0	0	0	5	0	1	0	0	1	0	0	0	0.639734	109.395

Figure 13 – PRISM Simulation for Vehicle Model Responding to Slope Increase (Level to Uphill)

2.3.3 Response to Slope Decrease (Uphill to Downhill)

Figure 14 shows the vehicle model's response to a decrease in slope (from uphill to downhill). Note that the engine rpm jumps up to 6452 when the slope first changes. Then the throttle decreases and the transmission upshifts (every other step) until engine rpm falls under 1800. Also, note that since the vehicle is travelling downhill, the position is changing by more than 1 meter at each step.

Step		Throttle...	DieselE...	rpm_m...	Transmission			Vehic...	TransmissionController			DriveTr...	terrain...	slopeM...	Rewards			
Action	#	throttle	DieselE...	rpm	Transm...	Transm...	Transm...	position	Transm...	Transm...	ctr	DriveTr...	theNext...	theNow...	"engine..."	"stall"	'mileage'	'vehicle...'
[tic]	207	60	0	1009	0	0	0	98	0	1	1	0	1	0	0	0	0.639734	109.395
[tic]	208	60	0	1009	0	0	0	99	0	1	0	0	1	0	0	0	0.639734	109.395
[tic]	209	60	0	1009	0	0	0	0	0	1	1	0	0	1	0	0	4.97277	850.343
[tic]	210	55	0	6452	0	0	0	7	0	2	0	0	0	1	0	0	4.78192	817.708
[tic]	211	50	0	5560	0	0	0	16	0	2	1	0	0	1	0	0	4.88145	834.727
[tic]	212	45	0	4960	0	0	0	24	0	3	0	0	0	1	0	0	4.78192	817.708
[tic]	213	40	0	4033	0	0	0	33	0	3	1	0	0	1	0	0	4.97277	850.343
[tic]	214	35	0	3433	0	0	0	40	0	4	0	0	0	1	0	0	4.97277	850.343
[tic]	215	30	0	2301	0	0	0	47	0	4	1	0	0	1	1	0	5.12621	876.582
[tic]	216	30	0	1701	0	0	0	52	0	4	0	0	0	1	1	0	5.12621	876.582
[tic]	217	30	0	1701	0	0	0	57	0	4	1	0	0	1	1	0	5.12621	876.582
[tic]	218	30	0	1701	0	0	0	62	0	4	0	0	0	1	1	0	5.12621	876.582
[tic]	219	30	0	1701	0	0	0	67	0	4	1	0	0	1	1	0	5.12621	876.582

Figure 14 – PRISM Simulation for Vehicle Model Responding to Slope Decrease (Uphill to Downhill)

2.3.4 Response to Drastic Slope Increase (Downhill to Steep Uphill)

Figure 15 shows the vehicle model's response to an increase in slope (from downhill to steep uphill). Note that the engine rpm drops to 0 when the slope first changes, and the vehicle stalls. Then the throttle increases and the transmission gear downshifts (every other step) until engine rpm becomes positive and the vehicle recovers from stall.

Step		Throttle...	DieselE...	rpm_m...	Transmission			Vehic...	TransmissionController			DriveTr...	terrain...	slopeM...	Rewards			
#		throttle	DieselE...	rpm	Transm...	Transm...	Transm...	position	Transm...	Transm...	ctr	DriveTr...	theNext...	theNow...	"engine..."	"stall"	'mileage'	'vehicle...'
224	30	0	1701	0	0	0	0	92	0	4	0	0	0	1	1	0	5.12621	876.582
225	30	0	1701	0	0	0	0	97	0	4	1	0	0	1	1	0	5.12621	876.582
226	30	0	1701	0	0	0	0	2	0	4	0	0	2	0	0	1	0	0
227	35	0	0	0	0	0	0	2	0	4	1	0	2	0	0	1	0	0
228	40	0	0	0	0	0	0	2	0	3	0	0	2	0	0	1	0	0
229	45	0	0	0	0	0	0	2	0	3	1	0	2	0	0	1	0	0
230	50	0	0	0	0	0	0	2	0	2	0	0	2	0	0	1	0	0
231	55	0	0	0	0	0	0	2	0	2	1	0	2	0	0	1	0	0
232	60	0	0	0	0	0	0	2	0	1	0	0	2	0	0	1	0	0
233	65	0	0	0	0	0	0	2	0	1	1	0	2	0	0	1	0	0
234	70	0	0	0	0	0	0	2	0	1	0	0	2	0	0	1	0	0
235	75	0	0	0	0	0	0	2	0	1	1	0	2	0	0	1	0	0
236	80	0	377	0	0	0	0	2	0	1	0	0	2	0	0	0	0.45928	78.5369
237	80	0	977	0	0	0	0	3	0	1	1	0	2	0	0	0	0.45928	78.5369
238	80	0	977	0	0	0	0	4	0	1	0	0	2	0	0	0	0.45928	78.5369
239	80	0	977	0	0	0	0	5	0	1	1	0	2	0	0	0	0.45928	78.5369

Figure 15 – PRISM Simulation for Vehicle Model Responding to Slope Increase (Downhill to Steep Uphill)

3 PRISM Terrain Modeler

The terrain modeler is a tool that automates the generation of probabilistic terrain models for PRISM from terrain specifications provided by the user. Terrain model specifications take the form of specialized PRISM comments that can be embedded in valid PRISM models. Running the terrain modeling tool on a PRISM model containing a terrain specification will result in a file containing the original PRISM model updated with an instance of the specified terrain model.

3.1 The Terrain Model

For the purpose of characterizing the terrain context for a ground vehicle, we employ a mathematical model that expresses the likelihood of traversing from one elevation to another. The tool supports two different terrain models, Kuchar and Max-Entropy.

3.1.1 Kuchar

The Kuchar model was derived in [1]. In that paper, Kuchar describes a Markov chain model for expressing probabilities of given changes in elevation. The probability of transitioning, from step n to step $n + 1$, from a starting altitude bin y_n to altitude bin y_{n+1} is given by Kuchar's equation (10):

$$p_{y_{n+1}, y_n}(n) = \int_{y_{n+1}-h/2}^{y_{n+1}+h/2} \frac{1}{\sqrt{2\pi\sigma^2(1-e^{-2\beta})}} \times e^{\frac{-(y-e^{-\beta}y_n)^2}{2\sigma^2(1-e^{-2\beta})}} dy$$

where h is the height of an altitude bin, and σ and β are autocorrelation parameters fitted to a database of actual terrain altitude samples obtained from the U.S. Geological Survey for the Great Plains and Rocky Mountain region between 102 and 112 degrees West longitude and 32 and 49 degrees North latitude. The autocorrelation parameters vary for different terrain types. Kuchar provides a table (Table 2) of values for the parameters for terrain categories of Smooth, Moderately smooth, Moderate, Moderately steep, and Steep.

The integrand in equation (10) is a probability density function (pdf) for a random variable that is normally distributed with mean $e^{-\beta}y_n$ and standard deviation $\sigma\sqrt{1-e^{-2\beta}}$.

3.1.2 Max Entropy

Essentially, the maximum entropy principle states that, when making inferences about a probability distribution based on incomplete information, one should arrive at the probability distribution that has maximum entropy among those permitted by the observed information. Jaynes [4] explains the motivation for this principle by showing that the permissible distributions are strongly concentrated near the one having maximum entropy – in other words, “distributions with appreciably lower entropy than the maximum permitted by our data are atypical of those allowed by the data.” For terrain elevation, it is clear that the absolute value of a change in elevation must be nonnegative; furthermore it is reasonable to assume we can obtain data on average elevation changes for a given region. For a

nonnegative random variable with a given expected value, the maximum entropy distribution is exponential:

$$p(z|F) = \frac{1}{F} e^{-z/F}$$

As before, we can derive the maximum likelihood estimate for F as follows. For ease of calculation we will work with the log likelihood, which, for a set of observations $z_i, i = 1, \dots, n$ is

$$\ln L(F|z) = \sum_{i=1}^n \ln \left(\frac{1}{F} e^{-z_i/F} \right) = \sum_{i=1}^n \left[\ln \left(\frac{1}{F} \right) - \frac{z_i}{F} \right]$$

Differentiating this with respect to F , and setting the derivative equal to zero implies

$$F = \frac{\sum_{i=1}^n z_i}{n}.$$

Given this model, with the relevant range of altitudes divided into bins of height h , the probability of an elevation change from the current bin to one k bins higher is given by

$$\frac{1}{2} \int_{(k-\frac{1}{2})h}^{(k+\frac{1}{2})h} \frac{1}{F} e^{-z/F} dz = \frac{1}{2} \left(e^{-(k-\frac{1}{2})h/F} - e^{-(k+\frac{1}{2})h/F} \right).$$

An analogous expression holds for the probability of moving down by k bins.

If the modelType parameter is set to maxEntropy and no F value is provided, the tool will compute F from the given (or default) Kuchar parameters in the following manner:

$$F = \sqrt{\frac{\sigma^2(1-e^{-2\beta})}{2}}.$$

3.2 Terrain Specifications

Conceptually the generated terrain model has one input (a Boolean value that indicates when the elevation may change) and two outputs (the current elevation and the current slope). Unfortunately, due to the lack of lexical scoping in PRISM, the user may need to provide suitable names for the inputs, outputs and state variables that constitute the terrain model to avoid name conflicts in the rest of their model.

Terrain specifications must be contained within a PRISM module named “terrainModel”:

```
module terrainModel
// .. Terrain Specification ..
endmodule
```

Terrain parameters are specified as whitespace delimited comments of the form:

```
// {keyword} {value}
```

Any text appearing after the {value} and before the next newline is ignored. The valid keywords, their role, types and default values and brief descriptions are provided in the following table.

Keyword	Role	Type	Default Value	Description
modelType	Model Selection	Either Kuchar or maxEntropy	Kuchar	Selects Kuchar or Max Entropy Model
F	Parameter	Double	0.0	Exponent in Max Entropy model
Beta	Parameter	Double	1.3e-3	Beta parameter in Kuchar model
Sigma	Parameter	Double	342.0	Sigma parameter in Kuchar model
maxElevation	Parameter	Double	100.0	Maximum elevation value
minElevation	Parameter	Double	0.0	Minimum elevation value
binCount	Parameter	Integer	10	Number of discrete elevations
tickName	Clock	Symbol	(empty)	Name of synchronizing event
currBinName	Internal State	Symbol (Integer)	currentElevationBin	Current discrete elevation value
nextBinName	Internal State	Symbol (Integer)	nextElevationBin	Next discrete elevation value
currElevationName	Output Formula	Symbol (Double)	currElevation	Current elevation
nextElevationName	Internal Formula	Symbol (Double)	nextElevation	Next elevation
currSlopeName	Output Formula	Symbol (Double)	currSlope	Current slope
updateElevationName	Input	Symbol (Boolean)	updateElevation	Enables transition from current to next elevation

Below is an example terrain specification:

```

module terrainModule
// modelType           Kuchar           Model Type
// F                   3.0               Max Entropy Exponent (unused)
// Beta                1.3e-3           Model parameter
// Sigma               342.0            Model parameter
// maxElevation        200.0            Upper bound on elevation
// minElevation        -200.0           Lower bound on elevation
// binCount            5                How many different elevations
// tickName            tick             Process synchronization
// currBinName          theNowBin        slopeModule state
// nextBinName          theNextBin       terrainModule state
// currElevationName    currentElevation current elevation signal
// nextElevationName    nextElevation    next elevation signal
// currSlope            currentSlope     slope signal
// updateElevationName  stepElevation    Name of the "step" signal
endmodule

```

Kuchar provides the following table to aide in choosing Sigma/Beta values based on a characterization of the terrain. Our default terrain model is classified as “moderate”.

Terrain Description	Sigma	Beta
Smooth	79.0	2.2e-3
Moderately Smooth	269.0	6.4e-4
Moderate (default)	342.0	1.3e-3
Moderately Steep	415.0	2.0e-3
Steep	1007.0	6.1e-4

3.3 Running the Tool

The tool is provided as a batch file called Terrain.bat (in Windows) or a shell script Terrain.sh (in Unix). Running the tool (with no arguments) opens a file chooser that allows the user to select the input file and the output destination. It is acceptable to replace the input file with the generated output.

The tool will remove any existing (previously generated) terrain model artifacts and replace them with the new terrain model. The artifacts removed by the tool include any code inside the body of the “terrainModule” (with the exception of the terrain specification comments), the entire “slopeModule”, and any other line outside of said modules preceded by the comment:

```
/// Generated Terrain Formula
```

This purging feature allows users to change the terrain specification, update the model, and analyze the results in an iterative fashion without undue editing. It also allows a user to develop a simple, initial PRISM model that parses without error in the PRISM tool and gets replaced in whole during the model generation process.

The tool generates a new body for the “terrainModule” and “slopeModule” as well as formulas for current elevation, next elevation, and current slope, each named by default or according to the terrain specification provided by the user.

The tool output for the example module described above is as follows:

```

module terrainModule
// modelType          Kuchar          Model Type
// F                  3.0              Max Entropy Exponent (unused)
// Beta               1.3e-3          Model parameter
// Sigma              342.0           Model parameter
// maxElevation        200.0           Upper bound on elevation
// minElevation        -200.0          Lower bound on elevation
// binCount            5              How many different elevations
// tickName            tick           Process synchronization
// currBinName          theNowBin      slopeModule state
// nextBinName          theNextBin     terrainModule state
// currElevationName    currentElevation current elevation signal
// nextElevationName    nextElevation  next elevation signal
// currSlope            currentSlope   slope signal
// updateElevationName  stepElevation  Name of the "step" signal
  theNextBin: [0..4];
  [tick] ((stepElevation) &
    (nextElevation <= -120.0)) ->
    0.0112: (theNextBin' = 1) + // -80.0
    0.9888: (theNextBin' = 0); // -160.0
  [tick] ((stepElevation) &
    (nextElevation > -120.0) &
    (nextElevation <= -40.0)) ->
    0.0107: (theNextBin' = 0) + // -160.0
    0.0110: (theNextBin' = 2) + // 0.0
    0.9783: (theNextBin' = 1); // -80.0
  [tick] ((stepElevation) &
    (nextElevation > -40.0) &
    (nextElevation <= 40.0)) ->
    0.0109: (theNextBin' = 1) + // -80.0
    0.0109: (theNextBin' = 3) + // 80.0
    0.9782: (theNextBin' = 2); // 0.0
  [tick] ((stepElevation) &
    (nextElevation > 40.0) &
    (nextElevation <= 120.0)) ->
    0.0110: (theNextBin' = 2) + // 0.0
    0.0107: (theNextBin' = 4) + // 160.0
    0.9783: (theNextBin' = 3); // 80.0
  [tick] ((stepElevation) &
    (nextElevation > 120.0)) ->
    0.0112: (theNextBin' = 3) + // 80.0
    0.9888: (theNextBin' = 4); // 160.0
  [tick] (! stepElevation) -> true;
endmodule

module slopeModule
  theNowBin: [0..4];
  [tick] ( stepElevation) -> (theNowBin' = theNextBin);
  [tick] (! stepElevation) -> true;
endmodule

/// Generated Terrain Formula
formula currentElevation = -160.0 + 80.0*theNowBin;
/// Generated Terrain Formula
formula nextElevation = -160.0 + 80.0*theNextBin;
/// Generated Terrain Formula
formula currentSlope = nextElevation - currentElevation;

```

4 Assume/Guarantee Analysis

Modern reasoning techniques and advances in computing technology have enabled tools capable of amazing feats of analysis. However, even the most capable tools are still limited by the size of the systems they can analyze. Compositional reasoning is a structured approach to verification that enables the analysis of systems whose size would otherwise overwhelm even modern analysis tools. Compositional reasoning enables the analysis of large systems of components by first decomposing the overall problem into several smaller and more tractable problems, solving those problems, and then combining the individual results to formulate a solution for the original problem.

Assume/guarantee reasoning is a compositional methodology that works well with a hierarchical design approach. The methodology relies on component contracts, which are assumptions on the component inputs that guarantee certain component behaviors. At each level of component hierarchy, analysis is limited to ensuring that the assumptions on the component inputs and the contracts of each of its sub-components are, together, sufficient to guarantee the desired component behaviors. Such correct by construction techniques minimize reasoning obligations and allow large, analyzable systems to be built from smaller, verified components.

4.1 Contracts

The assume/guarantee methodology requires the use of mathematically sound contracts. Contracts are a style of formal specification that enable one to capture, not only the expected component behavior, but the assumptions necessary in order to guarantee that behavior. To the component, the contract is an obligation. The component must (is obligated to) satisfy the contractual guarantees whenever the assumptions are met. To a system that employs the component, the contract is a guarantee. It can depend upon the component's contract (if the assumptions of that contract are satisfied), and it can, in turn, use the guarantees of that contract to satisfy its own contractual obligations. Contracts can be expressed mathematically as an implication: the assumptions imply the guarantee. Compositional reasoning follows from the compositional property of implication, which is to say, if A implies B and B implies C, then A implies C. The most crucial and easily overlooked step in this chain of reasoning is that that guarantee of the first contract must match (or satisfy) the assumption of the subsequent contract.

4.2 Typical Probabilistic Contracts

The statistical analysis of a component requires that certain assumptions be made about the component inputs and certain measurements be made on the component outputs. These assumptions and measurements (guarantees) constitute the component contract. Recall that the most crucial step in compositional reasoning is that the guarantee of one contract must match the assumption of another contract. Or, in its simplest form: a contract must guarantee the same sorts of things that it assumes.

When using tools such as PRISM, it is easy to express assumptions on inputs as probability distributions or Markov chains and it is natural to express properties as probabilities that certain events take place. The issue with such analysis is that the resulting contract is not compositional. In particular, the conclusion (that some event takes place with some probability) does not guarantee the same sorts of things that it assumes (that the inputs obey some given probability distribution or Markov relation).

Consequently, while typical probabilistic contracts may be quite useful for understanding various statistical behaviors of components, such contracts are of little use as part of an assume/guarantee methodology.

4.3 Compositional Probabilistic Contracts

There is a style of specification in PRISM that does work in a compositional assume/guarantee methodology. It requires the use of what are called Markov Decision Process, or MDP, models. In such models, the behavior of the input is essentially unconstrained. In each step of the model, an input may choose freely from any element of its domain. Assumptions in such models are expressed as simple probabilistic assertions about the behavior of the input (ie: the expected value of the input is 0.4) and properties are expressed as probabilistic assertions about the values of the outputs (i.e., the expected value of the output is 0.7). Such contracts are appealing in that they make the same sorts of assertions about the inputs as they do about the outputs. In other words, they are compositional and thus useful in an assume/guarantee methodology.

In order to verify such properties, PRISM must consider every possible sequence of inputs that satisfies the assumption (every sequence in which the expected value of the input is 0.4) and verify that the guarantee holds for every such sequence. If PRISM finds an input sequence for which the guarantee does not hold, the verification fails. Otherwise, the verification succeeds. Such analysis is called “adversarial analysis” because PRISM is searching for an adversary (a specific sequence of inputs) that causes the property to fail. If it cannot find such an adversary, it must concede that no such adversary exists and, thus, the contract will always hold.

4.4 Prototype Adversarial Assume/Guarantee Reasoning (AAGR) Framework

The PRISM tool supports adversarial analysis of MDP models of games: systems that have well defined end states. We are interested in reactive systems, such as vehicles operating over terrains, which do not have any particular end state. This lack of compositional reasoning support for reactive systems has compelled us to explore how such a reasoning system might be constructed.

The greatest challenge in the adversarial analysis of reactive systems is the size of the decision space. Recall that the adversarial analysis procedure must consider every possible sequence of inputs that satisfies the assumption. This means that the procedure must consider every possible probability assignment to every possible input state for every possible machine state. Considering that the size of the state space is exponential in the number of bits in the machine, it follows that the size of this decision space is doubly

exponential in the number of bits. For systems of any reasonable size, it would be impossible to enumerate all possible decisions before the heat death of the universe.

Previous research into adversarial analysis of games employed linear optimization to find the best adversary for a given system [6]. In other words, rather than considering all possible adversaries, it considered only the best (most malicious) adversaries. If no such adversary violates the contractual guarantee, the system must always satisfy the contract.

We attempt to leverage concepts from this previous work analyzing games and extend it to reasoning about reactive systems. The primary challenge with reactive systems is that properties are expressed in terms of expected values in the steady state, but the steady state of the machine depends upon the current decision. We were able to overcome this challenge by encoding the steady state of the system as an additional obligation to the linear optimization procedure. This solution provides a fast and effective decision procedure for deciding probabilistic assume/guarantee properties about reactive systems. In our experiments to date, even the largest problem we attempted was solved in only a few seconds.

4.5 The AAGR Prototype

In the AAGR prototype, component behavior is expressed in PRISM, properties are expressed in an Eclipse-based domain specific language called PrSL, and the adversarial analysis algorithm is implemented in MATLAB. We assume that the user is already familiar with model creation in the PRISM [8] probabilistic model checker.

4.5.1 Eclipse Workspace

Effective use of the AAGR framework requires that development be performed in an Eclipse workspace. This requirement is compatible with both PRISM and MATLAB and maximizes the utility of the PrSL tool. A “ready to use” example workspace containing four example projects has been provided with the AAGR distribution. When the tool is first started, it will query the user as to which workspace to use. Selecting the ready to use workspace provided with this distribution will allow for a quick evaluation of tool capabilities.

4.5.2 Probability Specification Language (PrSL)

The Probability Specification Language (PrSL) is a domain specific language for formulating properties to be checked by the AAGR tool. At the heart of it is an Eclipse-based editor that provides users with a modern specification environment complete with dynamic checking for syntax and well-formedness. In addition it provides a set of translations that tie behavioral descriptions written in PRISM with specifications written in PrSL and, ultimately, combines them both into assertions expressed in MATLAB.

PrSL specifications are composed of several sections. The interface section provides descriptions of the component’s various PRISM model principles, such as inputs, states and rewards. The equations section allows the user to construct descriptive shorthand notations for common specification expressions. Finally, the properties and refutation sections contain contract assertions that the framework will attempt to either verify or refute.

4.5.3 PrSL Interface Definition

The interface section is used to describe each principal found in the model. The interface section begins with the keyword “Interface” and contains a list of declarations that describe whether the principal is an input, state, constant or reward.

Each item in the interface list must contain an identifier that denotes the name of the principal, the type of principal (input, state, constant, or reward), and the position of this principal in the MATLAB reward vector. The following example shows an example of one of each type of principal.

```
Interface
    in           : input    @ 1;
    pin          : state    @ 2;
    pconst       : const    @ 3;
    reward1      : reward   @ 4;
```

During development, users are likely to add and remove inputs, states and rewards in their PRISM models. Also, the user is unlikely to know (or care) about the location of a given principal in the underlying MATLAB model. Consequently, users are discouraged from writing interface specifications by hand. Rather, PrSL provides a translation path that takes a PRISM model and generates an interface file. For a PRISM file named “foo.pm” the interface file will be named “fooInterface.prsl”. This machine generated interface file can then be imported into a hand-written specification, insulating the specification from changes in the PRISM model. Importing can be accomplished by using an import declaration. The following example illustrates the use of the import declaration.

```
Interface
    import "derivativeInterface.prsl";
```

4.5.4 PrSL Equations, Properties, and Refutations

Three sections are provided for users to specify properties of interest. The format of expressions in each of these sections is the same, but the manner in which expressions are interpreted in each section is different. The Equations section is a place for the user to place macros and other expressions that are frequently used. Any equation defined in this section can be referenced in expressions in the Properties and Refutations section. The Properties section is where the user places properties to be verified. Properties can refer directly to previously defined equations, or contain expressions that describe the property of interest. The Refutations section is where the user places properties expected to be false under some circumstances. Refutations and Properties are specified in the exact same syntax. Refutations, however, will be translated into MATLAB expressions that succeed when a counterexample is identified, rather than expressions that succeed when no such counterexample exists.

Here is an example of a small PrSL specification that employs each of these sections:

```
Interface
    import "derivativeInterface.prsl";

Equations
```

```

e1 = P pslope > 0;
e2 = P pslope > 0 & P pslope < 0.5 ;

Refutations
r1 = E pslope < 0;

Properties
p1 = E pslope < 3;

```

4.5.5 PrSL Expressions

The user may reason about principals using either an expected value operator (E) or, if the principal is Boolean, the probability operator (P). Principals may appear only in the context of one of these two operators. Note that expected value here means steady state (or long term) expected value. The expected value and probability operators may then be compared to constant bounds using any of the standard relational operations (<, <=, >, >=, ==, !=), resulting in a Boolean expression. The relational operators may also be used in ternary expressions, allowing the user to specify both upper and lower bounds on the expected value of a single principal (ie: $1.0 < E x < 2.0$). Such Boolean expressions may then be combined using standard logical operations to construct properties. The and (&), or (+), not (!), implies (->) and xor (#) logical operations are supported. The example below demonstrates the expression features of the language.

```

Interface
import "derivativeInterface.prsl";

Equations
e1 = P pslope > 0;
e2 = P pslope > 0 & P pslope < 0.5 ;
e3 = P reward1 < 0.85;

Refutations
r1 = e1;
r2 = E in < 2.3 ;
r3 = P in < 0.75 & P in < 0.9;

Properties
p1 = E pslope < 3;
p2 = P in > 0.75 + P in < 0.25 ;
p3 = e1 -> e2 ;
p4 = e1 # e3 ;

```

Validation passes are performed automatically to indicate to the user when an expression is incorrect or unsupported. For example, all relational operations must only compare a principal probability expression to a literal constant. If the user specifies something outside of that format, the tool will yield a graphical error and suggest to the user how to fix it. Shown below is an example of a few expressions that are not allowed.

```

Interface
import "derivativeInterface.prsl";

Equations
e1 = P pslope > P in;

```

```
e2 = pslope < 2 ;
e3 = e1 < e2 ;
```

```
Properties
r1 = e1;
```

4.5.6 MATLAB Output

For a given project, the final outputs of the PrSL tool are two MATLAB script files. The first script is the model file. It defines, among other things, the system transition matrix. This file is generated, along with the PrSL interface file, from the PRISM specification. For a PRISM file named “foo.pm” the model script will be named “fooModel.m”. The second script contains property assertions about the model. This script is generated from the PrSL specification. For a PrSL file named “foo.prsI”, the assertion script will be named “foo.m”. Performing an analysis involves first executing the model script and then executing the assertions script. The results of the analysis will then appear in the MATLAB console.

4.6 System Requirements

AAGR has been shown to work with the following software/systems:

- MATLAB 8.0.0.783 (R2012b) with Optimization Toolbox under Linux-64
- MATLAB 7.10.0.499 (R2010a) with Optimization Toolbox under WinXP-32

The PrSL executable was built using:

- Java 7
- Eclipse 4.2.1
- XText 2.3.1

In addition, PRISM 4.0.3 is assumed to be installed in the standard location, C:\Program Files\prism-4.0.3\.

4.7 Analyzing an AAGR Example

The distributed workspace provides four example projects. In this section, we focus on the /correlation project. This project comes with a PRISM module and a PrSL specification. In this section, we will give some background on this example and then walk thru the steps required to analyze it. The steps used to analyze this example are similar to the steps required to analyze any of the distributed examples.

4.8 The correlation model

The correlation model describes a system that detects temporal correlations in a binary input. Following is a PRISM description of its behavior.

```
module Correlation
s:[0..3];
[tic] (s=0) -> (s'= (input ? 1 : 0));
[tic] (s=1) -> (s'= (input ? 0 : 2));
[tic] (s=2) -> (s'= (input ? 3 : 0));
```

```

    [tic] (s=3) -> (s' = (input ? 0 : 2));
endmodule

formula alarm = ((s=2)|(s=3));

rewards "Alarm"
    alarm: 1;
endrewards

```

This system starts out in state 0, waiting for a true input. When it sees a true input it transitions to state 1. At this point, every time it sees an input that differs from the previous input, it advances (from 1 to 2, 2 to 3 and 3 to 4) until it reaches state 4 at which point it will oscillate between states 4 and 3 as long as the input keeps changing. If the machine ever sees the same input twice in a row, it returns to state 0. Note that we have defined a reward called “Alarm” that detects when the machine is in states 2 or 3. Our objective is to analyze the behavior of this system when the input is true 50% of the time.

First, let’s consider how we might ensure that our input is true 50% of the time. One possibility is to draw the input from a uniform probability distribution. The following module constrains the input in this manner.

```

module UniformInputModule
    input: bool;
    [tick] true -> 0.5:(input' = true) +
                0.5:(input' = false);
Endmodule

```

Now can we prove something about the expected value of the Alarm under this assumption? Can we prove, for example, that it is less than 0.5? In PRISM we would express that steady state property as follows:

$$R\{\text{"Alarm"}\} \leq 0.5 \quad [S]$$

Running the analysis in PRISM confirms this assertion. In fact, the computed probability of failure for this system is about .25 which is actually much better than .5. But now consider this: does this analysis prove that, any time the input is true with probability .5, the system will satisfy the contract? The answer is no. In particular, the sequence [T F T F T F T F..] (which is true with probability .5) will fail with probability 1.0. Conversely, the sequence [F F T T F F T T ..] fails with probability 0.0.

Unfortunately, the uniform distribution assumption that the input is true with probability 0.5 is not sufficient to guarantee the desired result: that the probability of failure is always less than 0.5. The problem with assumptions like uniform distributions is that they are too uniform: they don’t really account for temporal correlations in the input. In other words, they do not take into consideration worst case scenarios. This is where adversarial analysis comes in.

In AAGR, we employ PRISM's Markov Decision Models (mdp) to allow us to specify arbitrary inputs and then constrain the behavior of the inputs with explicit assumptions. The input specification for the correlation model now looks like this:

```
formula input = Ainput;
module AdverseInputModule
    Ainput: bool;
    [tic] true -> (Ainput' = true);
    [tic] true -> (Ainput' = false);
endmodule
```

Then start PrSL, point it to the provided workspace, and open up the correlation project. Click and drag the correlation.pm module into the editor pane. When the Prism model has focus (is the file being currently edited) the user can select “Translate from Prism” from the Translate menu. Figure 16 is a screenshot showing this action being performed for the file “correlation.pm” from the correlation project in the distributed examples.

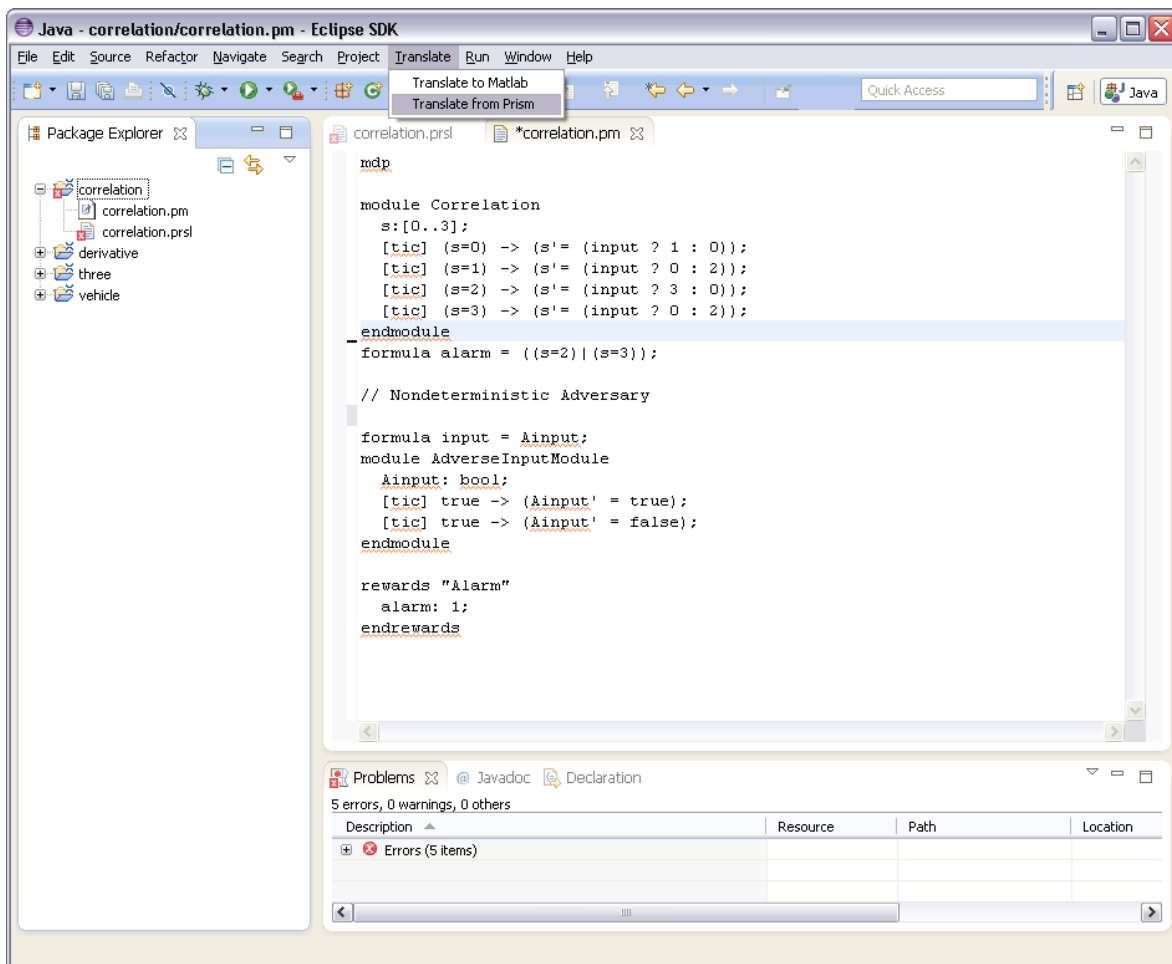


Figure 16 - Translate from PRISM menu

This will result in two new files in the project. The first is the MATLAB script that describes the behavior of the correlation model. The second is an interface file,

“correlationInterface.prsl”, that describes the principals of the correlation model. The interface file has the following contents:

```
Interface

Ainput      :  input      @  2;
s           :  state      @  1;
Alarm       :  reward     @  3;
```

Note that the input (Ainput), state (s) and reward (Alarm) values from the PRISM specification all appear here. They can now be used to express properties about this system. Note, too that the formula (alarm) is not provided here. If one needs to express a property or assumption about a formula, one must tie the formula to a reward.

We can now express our properties of interest using these principals. The provided PrSL file contains the following specifications.

```
Interface
    import "correlationInterface.prsl";

Equations

    // Property 1
    h1 = P Ainput == 0.5;
    r1 = P Alarm  > 0.0;

    // Property 2
    h2 = P Ainput == 0.5;
    r2 = P Alarm  < 1.0;

Refutations

    p1 = h1 => r1;
    p2 = h2 => r2;
```

We have specified two properties. The first (p1) says that, if the input is true with probability 0.5, the expected value of the alarm will always be greater than 0.0. The second (p2) says that if the input is true with probability 0.5, the expected value of the alarm must be less than 1.0. Both of these properties are expressed as “refutations” because, as it turns out, both are false.

After a specification has been created, the user can generate the Matlab script that will perform the analysis of the user specified properties. This can be done by invoking the “Translate to Matlab” command while editing the PrSL specification of interest. This is shown **Error! Reference source not found.** in Figure 17.

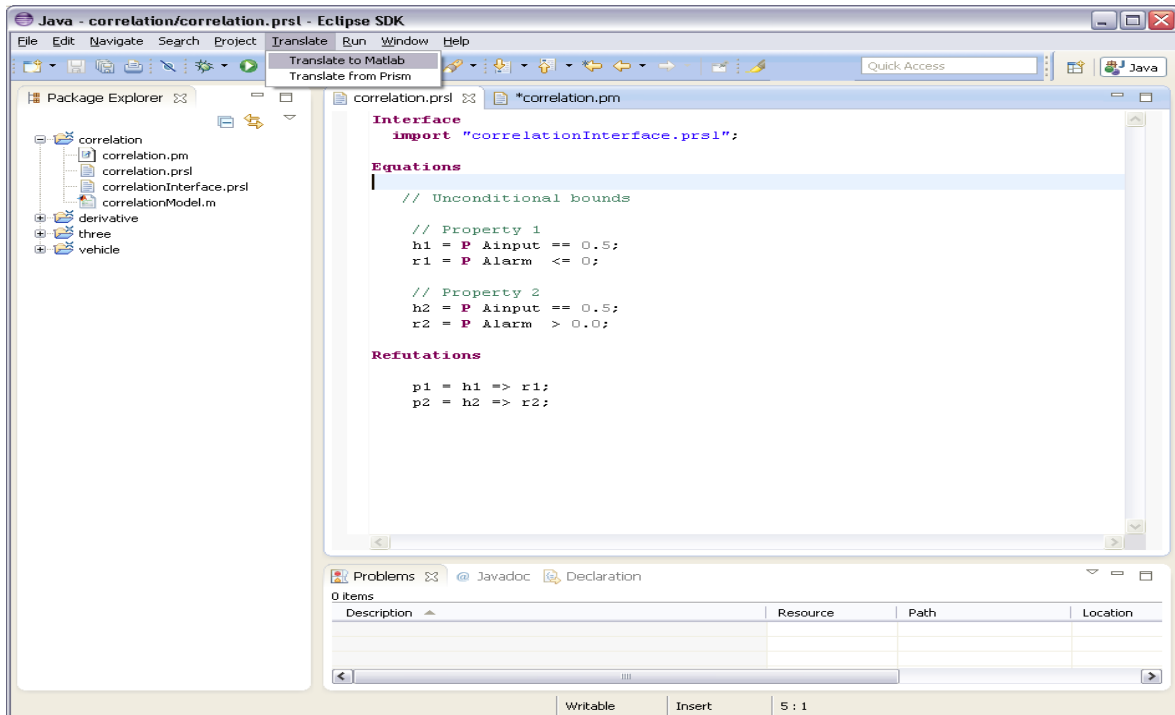


Figure 17 - Translate to MATLAB

This action produced one new file in the workspace, “correlation.m”. This is a MATLAB script that contains the assertions from the PrSL specification encoded as MATLAB commands.

To run the MATLAB scripts, start MATLAB and set the working directory to the correlation project in the distributed workspace. Note that your MATLAB path will need to include the /AAGR-MATLAB-source directory from the AAGR distribution so that MATLAB is able to find the “verify” and “refute” commands. Once the path and working directory have been set, the analysis proceeds by first executing the “correlationModel” script (to load the model into memory) and then executing the “correlation” script (to analyze the properties). Note that in this case, both properties were successfully refuted (Figure 18).

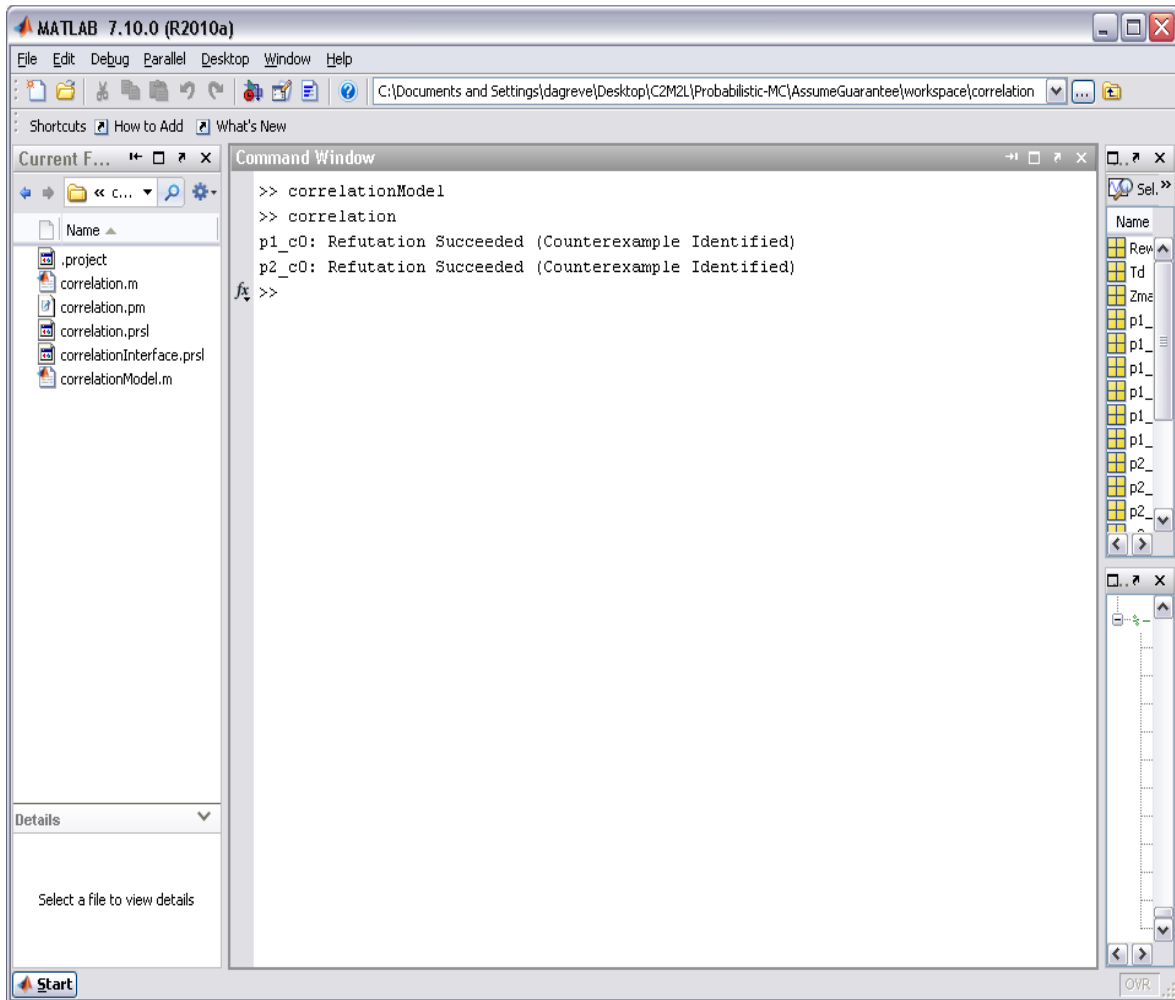


Figure 18 - Analyze example in MATLAB

4.9 Distributed Examples

In the distributed workspace we provide four projects containing different systems that have been used in developing and optimizing our analysis framework.

/correlation

Correlation is a system that detects temporal correlations in a binary input. This example clearly indicates the importance of adversarial analysis in establishing (or refuting) bounds on system behavior.

/three

Three is a somewhat non-linear system with three outputs. This system is intended as a simple regression test for the analysis algorithms. Note that not every clause of the second refutation in this example is falsifiable. Finding a counterexample to any clause of an assertion is enough to falsify the entire assertion. However, because each clause is processed separately, the tool currently reports those refutations as failures.

/derivative

Derivative is a small system that tracks the change in (derivative of) a 4-valued input. This system is intended as a small scale model that mimics some of the behaviors of the slope variable found in the larger vehicle model.

/vehicle

Vehicle is an early snapshot of the vehicle model. This model is particularly useful for identifying scalability issues in the algorithms. In our experience, this model is too big to load into 32-bit MATLAB. We were able to run it in 64-bit mode under Linux.

4.10 Future Work

There are several enhancements that can be made to the tool to make it more user-friendly. The issue of false negatives when processing multiple refutation clauses, for example, should be addressed. There are also some questions about what the most appropriate numerical error bounds should be. The current tool, for example, cannot decide questions with a precision greater than 0.0001. Also, a tighter integration with MATLAB could allow the user to drive the entire process from the Eclipse (PrSL) front end.

The issue of circular composition has vexed mathematicians for some time. Circular composition involves two contracts of the form $A \text{ implies } B$ and $B \text{ implies } A$. The question is: when can these contracts can be combined to soundly conclude that A and B are always true. There exist classes of temporal logic properties that admit circular composition [7]. Is there a similar class of probabilistic assertions? While general class of properties would be ideal, it could easily be the case that circular composition will always require an additional set of bootstrapping properties to preserve the soundness of composition.

Finally, an integrated infrastructure is needed to perform compositional verification and leverage assume/guarantee contracts. Such an infrastructure would provide comprehensive system analysis capabilities and enable the rapid evaluation of different design trade-offs and the impact of changes in environmental assumptions.

5 References

- [1] J. K. Kuchar, "Markov Model of Terrain for Evaluation of Ground Proximity Warning System Thresholds," *Journal of Guidance, Control, and Dynamics*, vol. 24, no. 3, pp. 428-435, May-June 2001.
- [2] T. Murphy, "100 MPG on Gasoline: Could We Really?," 17 July 2011. [Online]. Available: <http://physics.ucsd.edu/do-the-math/2011/07/100-mpg-on-gasoline/>. [Accessed 14 December 2012].
- [3] M. Ross, "Fuel Efficiency and the Physics of Automobiles," *Contemporary Physics*, vol. 38, no. 6, pp. 381-394, 1997.
- [4] E. T. Jaynes, "On the Rationale of Maximum-Entropy Methods," *Proceedings of the IEEE*, vol. 70, no. 9, p. 939, 1982.
- [5] Henzinger, Thomas A., Marius Minea, and Vinayak Prabhu. "Assume-Guarantee Reasoning." (2001).
- [6] Marta Z. Kwiatkowska, Gethin Norman, David Parker, Hongyang Qu: "Assume-Guarantee Verification for Probabilistic Systems", TACAS 2010: 23-37
- [7] John Rushby, "Formal Verification of McMillan's Compositional Assume-Guarantee Rule", CSL Technical Report, September 2001.
- [8] PRISM, <http://www.prismmodelchecker.org/>