Alloy-Guided Verification of Cooperative Autonomous Driving Behavior

by

MaryAnn Elizabeth VanValkenburg

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in Computer Science

by

May 2020

APPROVED:

Dr. Daniel J. Dougherty, Advisor

Dr. Craig A. Shue, Co-Advisor

Dr. Craig E. Wills, Head of Department

Abstract

Alloy is a lightweight formal modeling tool that generates instances of a software specification to check properties of the design. This work demonstrates the use of Alloy for the rapid development of autonomous vehicle driving protocols. We contribute two driving protocols: a Normal protocol that represents the unpredictable yet safe driving behavior of typical human drivers, and a Connected protocol that employs connected technology for cooperative autonomous driving. Using five properties that define safe and productive driving actions, we analyze the performance of our protocols in mixed traffic. Lightweight formal modeling is a valuable way to reason about driving protocols early in the development process because it can automate the checking of safety and productivity properties and prevent costly design flaws.

Acknowledgements

I would like to thank my husband, Art VanValkenburg, for his support during this project. I would also like to thank my advisors for their constant patience and kindness throughout the project. I daresay I enjoyed the experience.

Contents

List of Definitions

List of Figures

1 Introduction

Autonomous vehicles can prevent accidents that result from driver distraction and negligence. Connected technology such as the Vehicle-to-Vehicle communication system allows autonomous vehicles to not only predict and react to human drivers but to coordinate driving actions for improved road capacity and traffic flow.

Many obstacles stand in the way of the adoption of autonomous vehicles. In this work, we focus on the problem of ensuring the safety of autonomous vehicles in *mixed traffic* when they drive alongside human-driven vehicles.

Two competing goals for autonomous vehicles in mixed traffic are safety and productivity. Autonomous vehicles must drive defensively to account for the unpredictable nature of human drivers. However, safety alone is insufficient. A driving protocol that only protects safety may choose to prevent the car from moving at all. A driving protocol must allow the vehicle to make progress towards its destination.

Most efforts related to the development of autonomous driving protocols rely on simulation to demonstrate their protocols in action. Simulation can measure the performance driving protocols in complicated driving situations and can generate useful statistics about the predicted behavior of a protocol in real life. However, a simulation may not detect design flaws that only affect driving in rare situations. Formal modeling, or the use of mathematically rigorous tools to reason about a system, can be used to ensure that autonomous driving protocols are both safe and productive in all driving scenarios.

Safety and productivity are not the only goals in protocol design, yet safety and productivity alone dictate a sophisticated protocol. As the protocol becomes complicated, it becomes harder to reason about its correctness. In this project, we use the Alloy Analyzer to guide the development of a safe and productive driving protocol. This lightweight formal modeling tool allows for rapid specification and verification.

We define five properties for the assessment of driving policies. We refer to the assertion of these properties `possibleNextNotEmpty`, `noCollision`, `noCrossing`, `noDeadlock`, and `progress`. The `possibleNextNotEmpty` assertion checks that a driving policy does not exclude all possible courses of action. The `noCollision` and `noCrossing` assertions check that protocols are safe. The `noDeadlock` and `progress` assertions check that driving policies progress towards their destinations. Together, these properties demonstrate the competing needs for a protocol to be safe and productive.

We present four types of driving policies: `Oblivious`, `Paranoid`, `Normal`, and `Connected`. The `Oblivious` policy fails the `noCollision` assertion. The `Paranoid` policy fixes the flaw in the `Oblivious` policy that allowed collisions, but as a result, it fails the `noDeadlock` property. These two policies show the challenges in achieving both safety and productivity. The `Normal` policy describes typical human driving behavior. The `Connected` policy is the behavior of connected autonomous vehicles. These policies represent the various driving behaviors that are present in mixed traffic.

Chapter 2 presents current research into cooperative autonomous driving behaviors and approaches to testing these behaviors. We provide an overview of formal methods and the Alloy Analyzer. Chapter 3 shows how we used Alloy to model and analyze driving policies. Chapter 4 describes the driving policies we created and the evaluation criteria we developed to compare them. The chapter ends with a summary comparison of the different driving policies. Chapter 5 talks about key insights about modeling gained from analysis. We conclude that modeling complements simulation, especially at the development stage, and can be used in an automated way to check that a design is safe and productive.

2 Background

In this chapter, we provide an overview of the current research regarding connected autonomous vehicles. One challenge of introducing connected autonomous vehicles to the road is operating in mixed traffic with human-driven vehicles. Autonomous vehicles must be able to predict and react to human driving behavior. Most efforts to solve mixed traffic rely on simulations to demonstrate the safety of their proposed driving protocol. However, simulation is incapable of proving that the protocol works in every possible driving scenario. We suggest formal methods to ensure that driving protocols are safe in every scenario.

2.1 Cooperative autonomous driving

For the past 30 years, researchers have investigated dedicated short-range radio communication (DSRC) to improve the safety and efficiency of traffic [1]. Vehicles can use DSRC to communicate with each other through Vehicle-to-Vehicle (V2V) communication for safer and more efficient driving. Potential applications of V2V include emergency electronic brake light, in which the vehicle broadcasts an emergency-braking alert to nearby drivers, and intersection movement assist, which warns the driver when it is not safe to enter an intersection [2]. These connected applications have the potential to prevent accidents by promptly warning drivers of hazards. However, there is also interest in applying connected technology for cooperative autonomous driving.

Cooperative driving is the behavior of coordinating with other drivers for mutual gain. Examples of cooperative behavior include slowing down to allow other cars to merge, keeping to the leftmost lane of a traffic circle, and signaling to a driver at a four-way stop. *Cooperative autonomous driving* uses V2V to cooperate with other vehicles without needing input from the driver. The two main features of interest are autonomous lane changing and cooperative adaptive cruise control. In autonomous lane changing, vehicles negotiate merging order and accelerate or decelerate to provide space [3, 4, 5, 6]. In cooperative adaptive cruise control, vehicles rely on speed and location data from V2V rather than radar to match speed [7].

Cooperative autonomous driving can extend beyond the capacities of human drivers. One popular application is *platooning* [8, 9, 10, 11, 12]. Platooning involves vehicles driving at close distances to reduce wind drag and improve road capacity. Autonomous vehicles can drive in tight formation and at high speeds by continually communicating their location, car length, speed, and direction to the other members of the platoon over V2V. The platoon leader can broadcast an alert to all members of the platoon for a prompt reaction if the platoon needs to slow down or swerve to avoid obstacles.

Despite clear advantages to cooperative autonomous driving, the dependence of these behaviors on V2V raises concerns of privacy [13, 14, 15, 16], trust and integrity [17, 18, 19], and security [20, 21, 22, 23, 24]. Other concerns involve the safety of autonomous vehicles interacting with human drivers, known as *mixed traffic*. We focus on the issue of ensuring safety in mixed traffic.

2.1.1 Mixed traffic

It is unlikely that all drivers will upgrade to autonomous vehicles. For this reason, autonomous vehicles need to anticipate uncertainties involved with human drivers.

Efforts to improve autonomous vehicle driving protocols involve classifying observed driving situations according to their complexity to navigate [25, 26, 27], describing human driving behaviors [28], and using machine learning to predict the intentions of human drivers [29, 30]. Shladover [31], Zhou [7], and Navas [32] used simulation to demonstrate the performance degradation of cooperative autonomous features in mixed traffic.

Cooperative autonomous driving is a new research area, and few efforts have tested the safety of cooperative autonomous protocols in mixed traffic. All of the works mentioned so far used simulation to demonstrate their proposed protocols.

Simulation is the use of virtual models, hardware testbeds, or closed test tracks to test the design or implementation of a vehicle. Virtual simulations use computer models of traffic to test driving protocol performance. Virtual simulations work well for estimating response times of autonomous vehicles and improvements in traffic efficiency without the use of real vehicles. Hardware testbeds and closed test tracks use real vehicle components in secure test areas and are more expensive than virtual simulation but can provide additional assurance that the protocols will perform well in real traffic.

Amoozadeh uses virtual simulation to verify the design of a collision avoidance model [12]. Luo [5], Gao [29], and Wang [30] use virtual simulation and Hardware-in-the-loop testbeds to show the safety of their lane changing models. Vieira proposes a highly realistic simulation framework to verify the safety of autonomous protocols [33].

Simulation can show protocol designs working in complex traffic scenarios and can chance upon flaws in a design. However, simulation cannot prove that a protocol design is devoid of flaws. We investigated formal methods as a means to prove the correctness of autonomous driving protocols.

2.2 Formal methods for verification of design

Formal methods refers to a collection of mathematically rigorous techniques for exploring, and in some cases proving, the correctness of a design. Unlike simulation, formal methods do not require a coded implementation of the design. Formal methods test the properties of a design. By abstracting continuous data representations into finite values, they can prove the correctness of those properties.

There are two categories of formal methods: theorem provers and model checkers.

Theorem provers are tools for writing mathematical proofs. Interactive theorem provers such as Coq [34], PVS [35], and Isabelle [36] help the user to see what they have proved so far and how close they are to completing the proof. Of these, Coq and Isabelle can extract code from a proof development.

Theorem provers require user expertise to construct a proof of a property and cannot show why an attempted proof fails to be complete. If the user is unable to complete the proof, they have no way to tell whether they are not capable enough to complete the proof or that the result is not provable.

In contrast, model checkers can help the user see why a property might fail by showing counterexamples. *Model checkers* compile formally-written system specifications and then perform an automated search for counterexamples to user-specified properties about the system.

There are several existing model checkers with a range of applications. The symbolic model checker, nuXmv [37] is often used for hardware systems. Völker et. al. [38], used nuXmv to identify potential deadlock scenarios in an autonomous driving protocol. The work provided a method for analysis but did not apply the method to any specific automotive protocol. The SPIN model checker [39] is most often used for verifying software. One application of SPIN is Java Pathfinder, developed at NASA for the verification of spacecraft [40], which checks assertions about Java programs. TLA+ [41] has been used to model hardware and software. Notably, it was used by Intel for verification of their processor chip design [42].

Model checkers have push-button simplicity because they use an abstract version of the real system. Within the abstraction, they can prove things about the system. While they cannot conclude the correctness of the real system based on proof of the abstraction, model checkers are lightweight and have demonstrated effectiveness in uncovering subtle design flaws.

2.2.1 Alloy

Alloy [43] is a specification language and an analyzer for modeling software based on model checking. Because Alloy has a discrete representation of the design, it can perform an exhaustive analysis of software up to a given scope. Alloy presents counterexamples of asserted properties in a graphical format to help the user understand the design flaw.

Alloy natively supports more data types than SMV and supports structural properties better than SPIN and Java Pathfinder [44]. Notable applications of Alloy are by Zave to uncover concurrency bugs in the Chord distributed hash table protocol [45] and Svendsen [46] to verify safety and interoperability of train control protocols.

3 Approach

In the previous chapter, we explored research in the automotive industry related to new autonomous driving policies. We suggested formal modeling as a method to generate more persuasive statements of safety than simulation alone. We choose Alloy, a lightweight-formal modeling tool, to check that cars following connected autonomous driving policies are safe and productive even when intermingled with human-driven cars.

In this chapter, we introduce the abstraction of cars in segments of the road moving as a function of time. We define two simple driving policies, `Oblivious` and `Paranoid`. A car following an `Oblivious` policy can certainly have a collision, and a car following a `Paranoid` policy never will. The key result of this chapter is that Alloy *automatically* (i) discovers the non-safety of the `Oblivious` policy and shows the user a specific unsafe scenario, and (ii) exhaustively checks that there are no unsafe scenarios for `Paranoid` up to a given bound.

This automated analysis is crucial when reasoning about policies whose properties are not obvious to an informal human reasoner. Such policies are the topic of Chapter 4.

3.1 Alloy signatures, relations, and facts

The `sig` keyword, short for signature, is Alloy syntax to define a new type of object. Figure 3.1 is the Alloy specification for cars on the road. This specification shows three signatures: `Segment`, `Car`, and `Time`.

The `Segment` signature has two attributes, or *relations*, named row and lane. The row is labeled with positive integers such that a larger integer refers to a segment that is further ahead on the road. For simplicity, the road has two lanes, right and left. This specification could later evolve to a highway with more than two lanes.

The `Time` signature is also specified here. `Time` is a set of objects, which we will refer to

```
1  open util/ordering[Time] as trace
2
3  sig Time { }
4
5  pred exactlyPrecedes [pre, post: Time] {
6      post = pre.next
7  }
8
9  sig Segment {
10     lane: one Lanes,
11     row: one Int,
12 }
13
14 // Currently, just a two-lane road.
15 abstract sig Lanes {}
16 one sig Left extends Lanes {}
17 one sig Right extends Lanes {}
18
19 sig Car {
20     current: Segment one -> Time,  // Where the car currently is
21     possibleNext: Segment -> Time,  // Where the car can go next per policy
22 }
```

Figure 3.1: Alloy specification of `Segment`, `Car`, and `Time` signatures

in this text as *time points*. `Time` does not have any relations, but the `open` command on the top line specifies that time points are ordered. The `exactlyPrecedes` predicate specified on lines 5 to 7 says a time point, `pre` exactly precedes time point `post`, if `post` is the next element in the ordered set `Time`. This predicate is convenient when reasoning about two sequential time points. We explain predicates in detail later in this section.

The `Car` signature has two relations: `current` and `possibleNext`. The `current` relation specifies the `Segment` that the car occupies at a point in time. The `possibleNext` relation contains the set of segments that a car may occupy in the next point in time. For example, the `Segment` directly ahead of a `Car` may be included in `possibleNext` because it is within driving distance.

In simulation, one would expect positions on the road and time to be continuous values. In the Alloy specification, `Segment` and `Time` are discrete objects. Because Alloy has a discrete representation of the design, Alloy can complete an exhaustive search of the state space.

The keyword, `fact`, refers to properties of the system that are always true. Figure 3.2 shows three facts that add additional information about the `Car`, `Segment`, and `Time` signatures.

```
// Larger integer means further ahead on the road
fact rowMustBePositive {
    all r: Segment.row | r ≥ 0
}

// Uniqueness of segments
fact sameRowSameLaneImpliesSameSegment {
    all s1, s2: Segment |
        s1.row = s2.row and s1.lane = s2.lane implies s1 = s2
}

fact nextCurrentLocDerivedFromPossibleNext {
    all c: Car, t: Time |
        some t.next implies
            c.current.(t.next) in c.possibleNext.t
}
```

Figure 3.2: Supporting Alloy facts for Segment, Car, and Time

The first `fact` states that all segments must have positive row numbers. This statement has no real influence on the results of the modeling but ensures clarity when comparing rows. The second `fact` guarantees the uniqueness of segments by saying that segments with the same row and same lane must be the same segment. The last `fact` relates `current` with `possibleNext`. A car's `current` segment for the next sequential time point must be in `possibleNext`.

3.2 Alloy functions

The `fun` keyword, short for function, is a method for naming complex expressions. We use functions to filter the segments, relative to the `current` segment, that may be in `possibleNext`. We will soon explain how we use these filters to define driving policies.

Figure 3.3 shows how we model cars moving through space and time. The cycle shows the `possibleNext` and `current` segments from the `Car` represented in a table of values. In this report, we usually refer to the `possibleNext` and `current` relations as sets. However, the perspective of tables of values is useful in understanding the relationship between `possibleNext` and `current`.

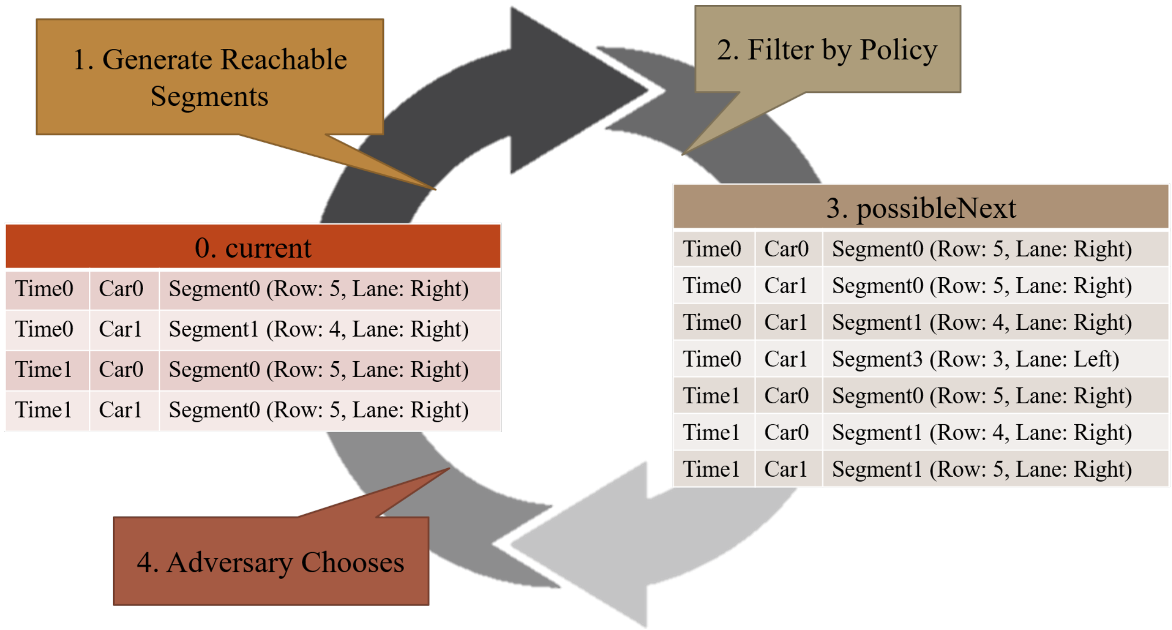


Figure 3.3: The `possibleNext` and `current` tables hold information about the location of cars. Driving policies determine the segments in `possibleNext` based on the car's current location. Alloy chooses a segment from `possibleNext` to be the car's next current location.

The `current` table, labeled in Figure 3.3 as Event 0, specifies which segment a car occupies in each time point. Cars can only occupy one segment at a time, so the `current` table has a 1-1-1 relation between `Time`, `Car`, and `Segment`. The `possibleNext` table, labeled as Event 3, contains the segments that the car might occupy in the next time point. It has a 1-1-many relation between `Time`, `Car`, and `Segment`. Each car, at any given point in time, can have many segments in `possibleNext`.

A *driving policy* is a combination of filters that determine the segments in `possibleNext`. In Figure 3.3, a driving policy is shown as two distinct events, labeled 1 and 2, to distinguish its dual action. In Event 1, Generate Reachable Segments, the driving policy filters out segments that are too physically distant from `current`. In Event 2, Filter by Policy, the policy filters, out of the remaining segments, those that it decides might cause a collision. In the Alloy specification, Events 1 and 2 occur simultaneously as an intersection of arbitrarily many filters.

Event 4, Adversary Chooses, represents the transition between the `possibleNext` segments in one time point and the `current` segment of the next time point. If there are multiple segments in `possibleNext` for a given car and time, one is chosen at random for the car to occupy next. When we assert properties of the policy, such as the `noCollision` property in Section 3.4.1, Alloy acts as an adversary and chooses segments from `possibleNext` that result in a collision for the next `current`. If Alloy fails to find counterexamples to the assertion, we can be confident that the property is true within the scope of the search.

3.3 Alloy predicates

An Alloy predicate, or `pred`, is a named constraint that can be applied to the model. Our driving policies are defined using predicates and formulated by combining filters. Another way we use the Alloy `pred` is to specify properties about our model, such as the `noCollision` property, which we will discuss later in this section.

3.3.1 Driving Policy: `Oblivious`

Figure 3.4 shows a simple driving policy with one filter on `possibleNext`.

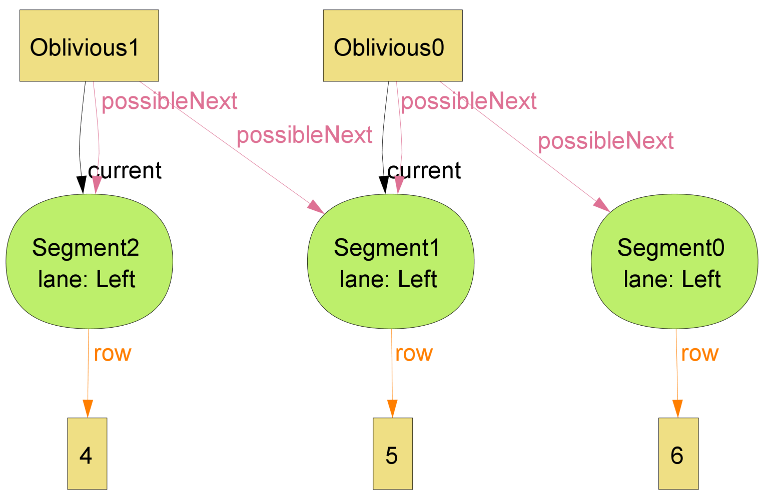
```
1  // RULE
2  fun ForeDiagOrStop (c: Car, t: Time) : set Segment {
3      // + := set union
4      fore[c, t] + diag[c, t] + here[c, t]
5  }
6
7  sig Oblivious extends Car {}
8
9  pred ObliviousPolicy [c: Car, t: Time] {
10     c.possibleNext.t = ForeDiagOrStop[c, t]
11     c in Oblivious
12 }
```

Figure 3.4: Specification of `ForeDiagOrStop` filter and the `Oblivious` driving policy

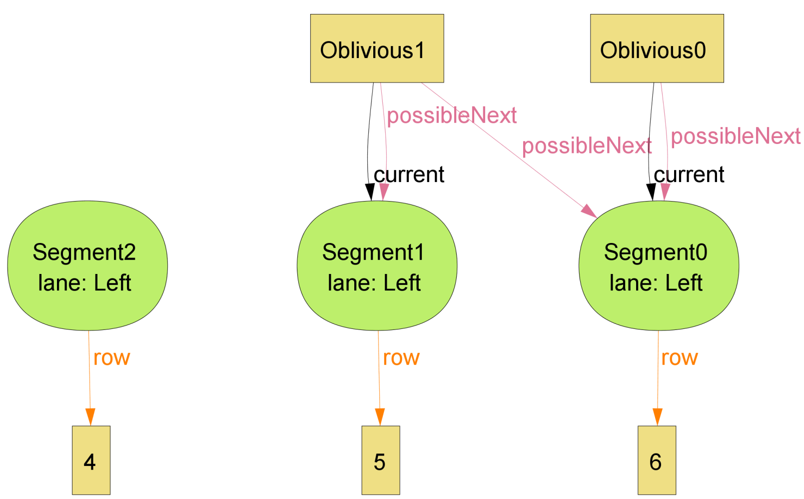
The function `ForeDiagOrStop` on line 2 of Figure 3.4 is a filter that determines which segments are within driving reach of the car. The function takes, as input, a car and a time point and returns, as output, the set of segments that are within reach of the car at the time point. The `fore`, `diag`, and `here` keywords are helper functions that find the segments in front of, diagonal to, and currently occupied by the car. The helper functions are shown in full in Appendix C.

Definition 1 (The `ForeDiagOrStop` filter) *For a given `Car` and `Time`, the filter `ForeDiagOrStop` returns the segments that are in front of, diagonal to, or currently occupied by the car.*

Definition 2 (The `Oblivious` policy) *For a given `Car` and `Time`, the `Oblivious` policy defines `possibleNext` as the segments that are returned by the `ForeDiagOrStop` filter.*



(a) At Time0, Oblivious0 is in Segment1 and Oblivious1 is in Segment2.



(b) At Time1, Oblivious0 moves into Segment0 and Oblivious1 moves into Segment1.

Figure 3.5: Instance of all cars following `Oblivious` policy with no collisions

Figure 3.5 is an Alloy-generated instance of the `Oblivious` policy. The figure shows two separate diagrams projected over `Time`; but together, these diagrams comprise one Alloy-generated instance.

In the first time point, the two cars occupy different segments. In the next time point, both cars move. Oblivious0 moves forward from the left lane of row 5 to the left lane of row 6. Oblivious1 moves from the left lane of row 4 to the left lane of row 5.

The instance in Figure 3.5 shows two cars safely driving no the road. However, our goal is to check that the `Oblivious` policy prevents cars from colliding with each other. For this, we write an assertion of a safety property.

3.4 Alloy assertions

An assertion, or `assert`, is a claim about the specification. Alloy generates instances of the specification to check that the assertion holds. If the assertion is false, Alloy finds counterexamples and displays them to the user.

3.4.1 The `noCollision` property

Definition 3 (The `noCollision` property) *A collision occurs when two cars occupy the same segment at the same time. A time point has the* `noCollision` *property when no two cars occupy the same segment.*

Figure 3.6 shows the assertion of the `noCollision` property for cars following the `Oblivious` policy. The predicate on line 1 specifies that, for a specified time point, all cars follow the `Oblivious` policy. The assertion on lines 5 through 13 states, "for any two sequential points in time, when all cars follow the `Oblivious` policy, if cars start in a state of no collision then they will end in a state of no collision."

The assertion requires that cars originate in a state of no collision. Without this constraint, Alloy would present counterexamples in which cars start in a state of collision.

```
1  pred AllOblivious [t: Time] {
2      all c: Car | ObliviousPolicy[c, t]
3  }
4
5  assert AOnoCollision {
6      all pre, post: Time |
7      (
8          exactlyPrecedes[pre, post] and
9          noCollision[pre] and
10         AllOblivious[pre]
11     )
12     implies noCollision[post]
13 }
14 check AOnoCollision for 5 but 2 Time
```

Figure 3.6: Checking `noCollision` property when cars follow the `Oblivious` policy

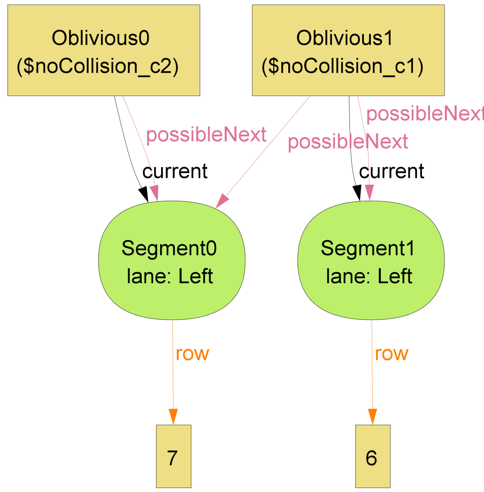
These counterexamples are uninformative of faults within the driving policy. However, if the `noCollision` property is maintained after cars follow the driving policy, then the driving policy is sufficient to prevent collisions.

Line 14 of Figure 3.6 is a command for Alloy to check the truthfulness of the assertion. The value 5 is the scope for which Alloy should check all instances. A scope of 5 includes all instances that employ up to 5 Segments and 5 Cars. The scope for Time is 2 because only two sequential time points, `pre` and `post`, are needed to check the behavior of the policy.

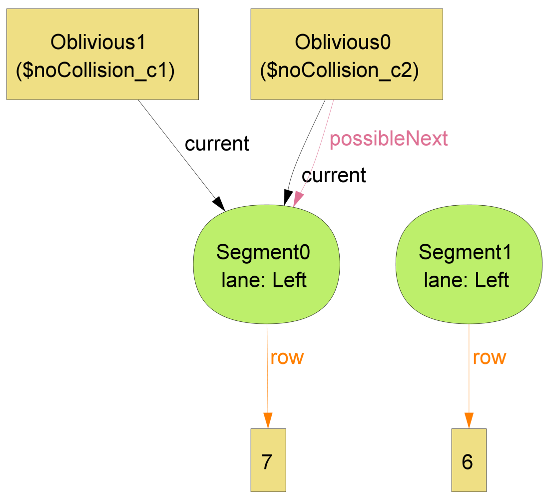
Alloy guarantees the truthfulness of assertions up to the specified scope. Increasing the scope increases the confidence that the assertions are true in all cases. Increasing the scope also increases the number of instances Alloy checks and is exponentially complex. However, the *small-scope hypothesis* says that most, if not all, flaws in the specification are evident in a small scope. For all of the analyses in this work, the scope is below 7.

3.5 Analysis of the `Oblivious` and `Paranoid` policies

Figure 3.7 shows an Alloy-generated counterexample to the `noCollision` assertion when cars follow the `Oblivious` policy. Oblivious0 and Oblivious1 start in adjacent rows. Oblivious1 then changes lanes and collides with the stopped Oblivious0.



(a) At Time0, Oblivious0 is in Segment0 and Oblivious1 is in Segment1.



(b) At Time1, Oblivious1 changes lanes into Segment0 and collides with Oblivious0.

Figure 3.7: Counterexample to the `noCollision` assertion when all cars follow the `Oblivious` policy

The `Oblivious` policy did not prevent collisions. After examining the counterexamples, we hypothesize that we can make the policy safe by restricting `possibleNext` segments to those that no other car can reach.

3.5.1 Driving Policy: `Paranoid`

The `Paranoid` policy, shown in Figure 3.8, includes the `ForeDiagOrStop` filter present in the `Oblivious` policy. This filter keeps the segments in front of, diagonal to, or currently

```
sig Paranoid extends Car {}

// RULE
fun AvoidForeDiagOrStopOfPeerExceptSelf (c: Car, t: Time) : set Segment {
    // The set of segments where for each segment s, s is not physically
    // reachable by another car
    {s: Segment | all peer: Car-c |
        s in c.current.t or
        s not in ForeDiagOrStop[peer, t]}
}

pred ParanoidPolicy [c: Car, t: Time] {
    // & := set intersection
    c.possibleNext.t = ForeDiagOrStop[c, t] &
                        AvoidForeDiagOrStopOfPeerExceptSelf[c, t]
    c in Paranoid
}
```

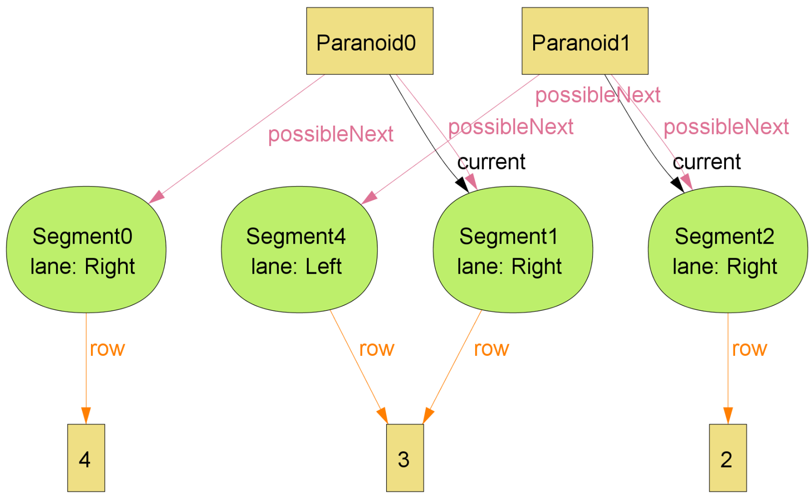
Figure 3.8: Driving Policy: `Paranoid`

occupied by the car. The `Paranoid` policy has an additional filter, `AvoidForeDiagOrStopOfPeerExceptSelf`, that excludes segments that are in front of, diagonal to, or currently occupied by another car.

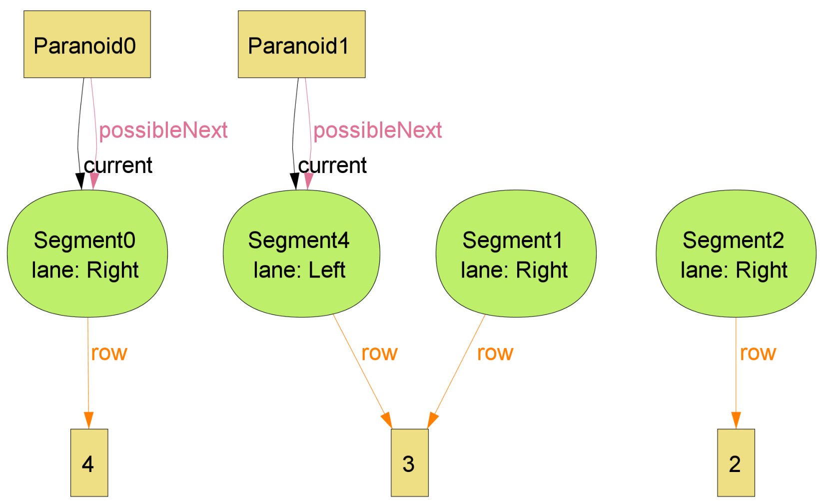
Definition 4 (The `AvoidForeDiagOrStopOfPeerExceptSelf` filter) *For a given `Car` and `Time`, the filter `AvoidForeDiagOrStopOfPeerExceptSelf` returns the `current` segment. The filter also returns all other segments that are **not** in front of, diagonal to, or currently occupied by another car.*

Definition 5 (The `Paranoid` policy) *For a given `Car` and `Time`, the `Paranoid` policy defines `possibleNext` as the segments that are returned by both the `ForeDiagOrStop` and `AvoidForeDiagOrStopOfPeerExceptSelf` filters.*

With the inclusion of the `AvoidForeDiagOrStopOfPeerExceptSelf` filter, Alloy finds no counterexamples to the `noCollision` assertion. Figure 3.9 shows an instance of cars following the `Paranoid` policy with no collisions.



(a) At Time0, Paranoid0 is in Segment1, and Paranoid1 is in Segment2.



(b) At Time1, Paranoid0 moves forward into Segment0, and Paranoid1 changes lanes into Segment4.

Figure 3.9: Instance of all cars following `Paranoid` policy with no collisions

The `Paranoid` policy is safe from collisions. However, safety is not the only requirement of a driving protocol. The `Paranoid` policy also needs to be productive.

3.5.2 The `noDeadlock` assertion

A driving protocol is productive if it allows cars to reach their destinations. One way a protocol can be unproductive is when deadlocks occur. A *deadlock* is a scenario in which each car waits for the other to act before deciding its own course of action. Autonomous vehicles following the same driving protocol are susceptible to deadlocks. Without a tiebreaking procedure, neither car is able to move.

Definition 6 (The `noDeadlock` property) *A deadlock occurs when the only segment in `possibleNext` is the `current` segment. A time point has the `noDeadlock` property if at least one car has a segment other than `current` in `possibleNext`.*

Figure 3.10 shows the assertion of `noDeadlock` applied to the scenario where all cars follow the `Paranoid` policy. The assertion is interpreted as, "at any point in time, if no cars are colliding and all cars are following the `Paranoid` policy, then we assert that there exists a car that has a segment other than the current segment in `possibleNext`."

```
assert APnoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllParanoid[t]
    )
    implies noDeadlock[t]
}
check APnoDeadlock for 5 but 1 Time
```

Figure 3.10: Checking the `noDeadlock` assertion when all cars follow `Paranoid` policy

Alloy finds a counterexample, shown in Figure 3.11, to the `noDeadlock` assertion. Two cars start in adjacent segments. Since both cars have the ability to move into Segment2, the `Paranoid` policy excludes Segment2 from each car's `possibleNext`.

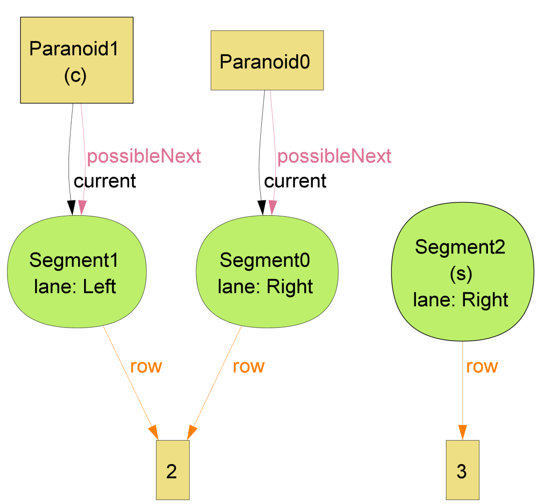


Figure 3.11: Counterexample to `noDeadlock` assertion when all cars follow `Paranoid` policy

The behavior shown in Figure 3.11 is similar to the situation in which cars from two lanes try to merge into one lane. Without clear and timely cues, both drivers may slow down to prevent colliding with the other car and neither will proceed.

The `Paranoid` policy is not aware of the segments in the other car's `possibleNext`. The policy filters segments that are observed to be within the immediate reach of the other car. In Chapter 4, we create a Connected policy in which connected autonomous vehicles broadcast their `possibleNext` tables to each other. Sharing `possibleNext` represents the information flow afforded by V2V. By sharing their intentions with each other, connected autonomous vehicles can avoid deadlocks.

In subsequent analyses, we add a predicate to the `noDeadlock` assertion that says that there exists some vacant segment ahead on the road. This rules out the counterexample in which cars are in *gridlock*, the scenario in which the road is saturated with vehicles. The gridlock counterexample is uninformative; without vacant segments, no driving policy can resolve a gridlock. With this additional vacancy criterion, we ensure the counterexamples to the `noDeadlock` assertion result from the driving policy.

3.6 Analysis of mixed traffic

So far, we have considered scenarios in which all cars on the road follow the same policy. We can easily specify a scenario in which cars follow either the `Oblivious` or the `Paranoid` policy. We show the specification of this scenario in Figure 3.12.

```
pred MixedObliviousOrParanoid [t: Time] {
    all c: Car | ObliviousPolicy[c, t] or ParanoidPolicy[c, t]
}
```

Figure 3.12: Specification of mixed traffic: `Oblivious` and `Paranoid`

This predicate allows for mixed traffic with any proportion of `Oblivious` or `Paranoid` policy-following cars. This predicate includes the scenarios in which all cars follow the same policy. The mixed traffic scenario inherits the flaws of both policies. This mixture of `Oblivious` and `Paranoid` policy-following cars fails the `noCollision` and `noDeadlock` assertions. In the next chapter, we show two policies that are safe when all cars follow the same policy but result in collisions when cars follow different policies.

Autonomous driving policies must be safe and productive. In this Alloy specification, policies are safe and productive if the `noCollision` and `noDeadlock` assertions are true. The `Oblivious` policy failed the `noCollision` assertion by taking no action to check for nearby cars. The `Paranoid` policy, with the addition of the `AvoidForeDiagOrStopOfPeerExceptSelf` filter, passed the `noCollision` assertion, but it failed the `noDeadlock` assertion.

In the next chapter, we create driving policies composed of multiple filters such that safety and productivity are not as obvious. We also introduce three more properties to check for safety or productivity. Even in complex driving scenarios, Alloy automatically checks for counterexamples of these properties. We use Alloy-generated counterexamples to inform the development of driving protocols. By checking that driving policies ensure all five properties, we gain confidence that our driving policy will perform well on the road.

4 Results

In the last chapter, we used the `Oblivious` and `Paranoid` policies to introduce analysis using Alloy and to demonstrate the assertion of the `noCollision` and `noDeadlock` properties. In this chapter, we describe `Normal` and `Connected` driving policies. Two versions of the `Normal` policy, `NormalAvoid` and `NormalAvoidLaneChange`, represent the typical driving behavior of non-autonomous, human-operated vehicles at different levels of complexity. The `Connected` policy represents the behavior of a connected autonomous vehicle. We show four iterations of the `Connected` policy, numbered from I to IV, to demonstrate Alloy-guided design. `ConnectedIV` is the most advanced policy, and it meets all of our specified safety and productivity goals in mixed traffic.

In addition to the `noCollision` and `noDeadlock` properties defined in Chapter 3, we introduce the `possibleNextNotEmpty`, `noCrossing`, and `progress` properties.

The `possibleNextNotEmpty` property requires that the filters that compose a driving policy are not mutually exclusive. The `noCrossing` property refers to the type of collision in which cars in adjacent lanes attempt to change lanes at the same time. The `progress` property is a stronger statement than `noDeadlock` and says at least one car will move to a different segment in the next time point. These five properties employ different techniques for checking the safety and productivity of policies.

Section 4.3 summarizes the results of analyzing each driving policy with all five properties. Section 4.4 reviews key takeaways from this modeling. Chapter 5 goes into further detail about modeling with Alloy.

4.1 `Normal` driving policies

To compare the safety of a connected autonomous vehicle in mixed traffic, we invent a policy that describes human drivers. Because it represents the baseline behavior of modern drivers, we refer to it as the `Normal` policy. The `Normal` policy has two requirements: it needs to be somewhat unpredictable to represent the variability of human driving, and it needs to be safe when all cars follow it to not confound the property assertions of mixed traffic.

It is difficult to specify how humans drive. (See Moridpour's review for an extensive taxonomy of lane-changing behaviors including psychological models of human drivers [3].) Rather than define the behavior of human drivers, the `Normal` policy allows for unpredictable maneuvers. The `Oblivious` policy described in Chapter 3 allowed for multiple segments in `possibleNext`, and the Alloy analyzer chose one at random to be the next `current` segment. The `Normal` policy, like the `Oblivious` policy, allows multiple segments to be in `possibleNext`. The randomness of Alloy's selection is used to represent the difficulty of predicting a driver's behavior.

Mixed traffic is susceptible to the flaws of the individual policies. To assess the safety of `Connected` policies mixed traffic, the `Normal` policy needs to be safe in homogeneous traffic. If the `Normal` policy is safe, then counterexamples are informative of the interaction between the different policies.

One of the counterexamples to the `noCollision` assertion of the `Oblivious` policy was two adjacent cars moving forward and diagonally into the same segment. The `noCollision` assertion failed because the cars shared a segment in `possibleNext`. However, each car's `possibleNext` also contained segments that would not have resulted in a collision. The `Normal` policy ensures the uniqueness of segments in `possibleNext` by using observable information about other cars to avoid the segments they may occupy next. Unlike the `Paranoid` policy, the `Normal` policy passes the `noDeadlock` assertion.

We used two `Normal` policies to assess the `Connected` policy. In the first policy, `NormalAvoid`, cars may move forward or stop, but they do not move to a segment that is currently occupied by another car. This version excludes lane changing maneuvers for simplicity. The second policy, `NormalAvoidLaneChange`, allows cars to move to the diagonal lane, but only if the segment next to the car is vacant. When all cars follow the same policy, both `Normal` policies prevent collisions.

4.1.1 The `possibleNextNotEmpty` property

One danger of using multiple filters to define a policy is that the filters might exclude all segments. If this is the case, `possibleNext` may be an empty set. This phenomenon does not translate to the physical world. In the real world, an empty `possibleNext` would mean cars cannot move forward nor can they remain stopped. Their only course of action is to cease to exist.

Thus, for the `current` and `possibleNext` model to work properly, `possibleNext` must include at least one segment for each time point. It may be that `possibleNext` only contains the `current` segment. If that is the case, the car will remain in place for the next time point. For both of the `Normal` policies, the `possibleNextNotEmpty` assertion is true.

Definition 7 (The `possibleNextNotEmpty` property) *A time point has the property `possibleNextNotEmpty` if all cars have some segment in `possibleNext`.*

4.1.2 Driving Policy: `NormalAvoid`

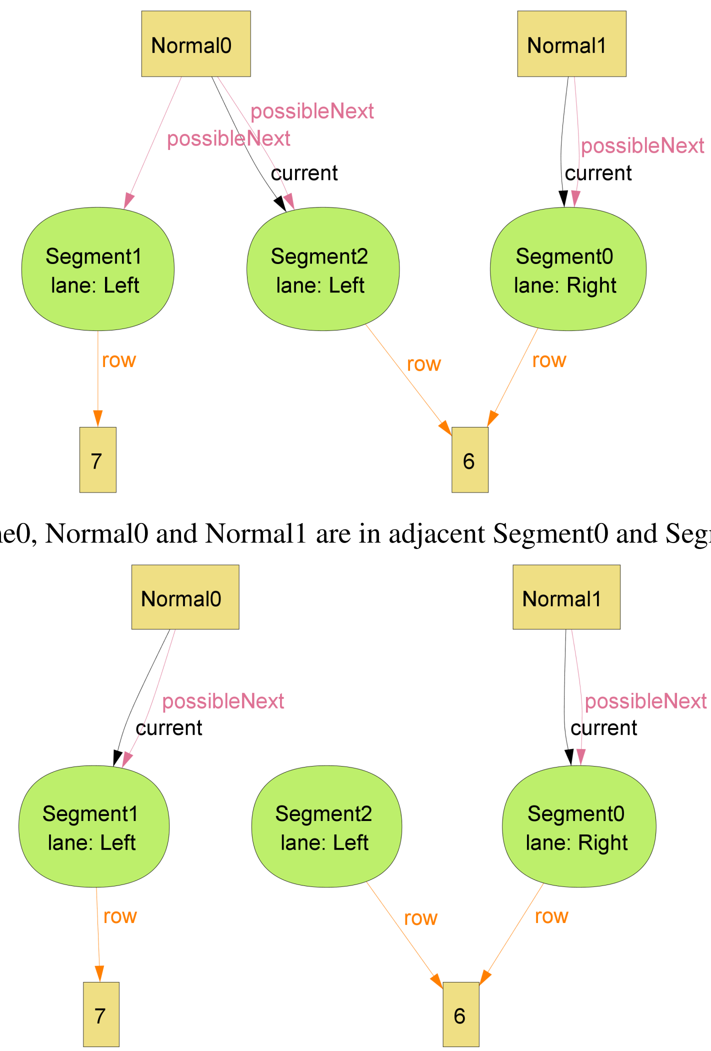
The `NormalAvoid` policy is comprised of two filters: `ForeOrStop` and `AvoidOccupiedExceptSelf`.

Definition 8 (The `ForeOrStop` filter) *For a given `Car` and `Time`, the filter `ForeOrStop` returns the segments that are in front of or currently occupied by the car.*

Definition 9 (The `AvoidOccupiedExceptSelf` filter) *For a given `Car` and `Time`, the filter `AvoidOccupiedExceptSelf` returns the `current` segment. The filter also returns all other segments that are not in another car's `current`.*

Definition 10 (The `NormalAvoid` policy) *For a given `Car` and `Time`, the `NormalAvoid` policy defines `possibleNext` as the segments that are returned by both the `ForeOrStop` and `AvoidOccupiedExceptSelf` filters.*

The intersection of `ForeOrStop` and `AvoidOccupiedExceptSelf` is always at least one segment, `current`, and at most two segments, `current` and the forward segment. The forward segment may be excluded because it does not exist in the Alloy instance or because it is currently occupied by another car. The `NormalAvoid` policy satisfies the assertion `possibleNextNotEmpty`.



(a) At Time0, Normal0 and Normal1 are in adjacent Segment0 and Segment2.

(b) At Time1, Normal0 moves forward into Segment1 and Normal1 remains in Segment0.

Figure 4.1: Instance of all cars following the `NormalAvoid` policy with no collisions

Alloy found no counterexamples to the `noCollision` assertion when all cars follow the `NormalAvoid` policy. Figure 4.1 shows an instance of cars following the `NormalAvoid` policy without collisions.

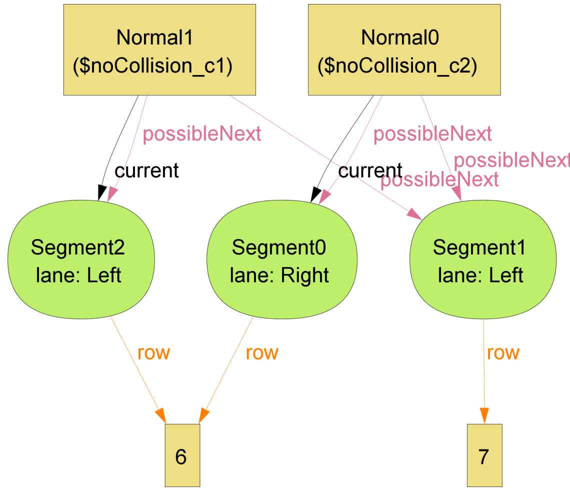
Unlike the `Paranoid` policy, cars following the `NormalAvoid` policy cannot change lanes. This allows them to safely move past each other, so the `NormalAvoid` policy passes the `noDeadlock` assertion.

The `NormalAvoid` policy is a representation of stop-and-go traffic. In stop-and-go traffic, the safest course of action is to remain in the same lane. Changing lanes does not significantly improve travel time, and it incurs additional risk of collision. As long as all cars stay in their lanes, they need not worry about other cars merging in front of them.

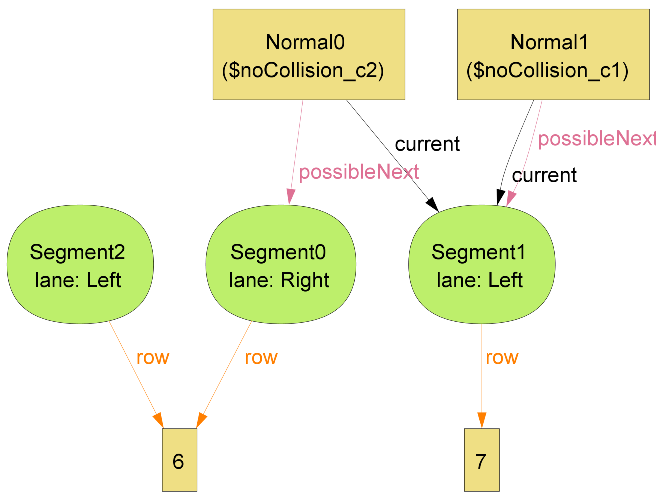
In reality, cars in stop-and-go traffic still sometimes collide due to sudden braking or driver distraction. These accidents are due to human error and not faulty policies, so we exclude them from analysis.

4.1.3 Driving Policy: `NormalAvoidLaneChange`

To introduce lane-changing capabilities, the `ForeOrStop` filter is replaced with `ForeDiagOrStop`. With this change, Alloy finds a counterexample to the `noCollision` assertion. Figure 4.2 shows two adjacent cars. One car attempts to move forward into a vacant segment. The other car attempts to move diagonally into the same vacant segment.



(a) At Time0, Normal0 is in Segment0 and Normal1 is in Segment2.



(b) At Time1, both Normal0 and Normal1 move into Segment1.

Figure 4.2: Counterexample to `noCollision` when all cars follow the `NormalAvoid` policy with the addition of the diagonal segment

This counterexample is similar to when a driver "cuts off" another driver by merging into their lane and causing them to slow down. In this scenario, the merging driver is responsible for checking that the lane is empty before attempting to merge. We represent this lane check with the `AvoidDiagonalIfAdjacentOccupied` filter.

Definition 11 (The `AvoidDiagonalIfAdjacentOccupied` filter) *For a given `Car` and `Time`, if the adjacent segment to the `Car` is occupied, the filter `AvoidDiagonalIfAdjacentOccupied` excludes the diagonal segment. The filter `AvoidDiagonalIfAdjacentOccupied` returns all* *other segments.*

Definition 12 (The `NormalAvoidLaneChange` policy) *For a given `Car` and `Time`, the `NormalAvoidLaneChange` policy defines `possibleNext` as the segments that are returned by the conjunction of the `ForeDiagOrStop`, `AvoidOccupiedExceptSelf`, and `AvoidDiagonalIfAdjacentOccupied` filters.*

If the segment next to the car is occupied, `AvoidDiagonalIfAdjacentOccupied` filters out the diagonal segment, reasoning that the adjacent car will most likely move forward. This would be like the human driver checking to see if there is a car beside them. With the `AvoidDiagonalIfAdjacentOccupied` filter, the `NormalAvoidLaneChange` policy passes the `noCollision` assertion.

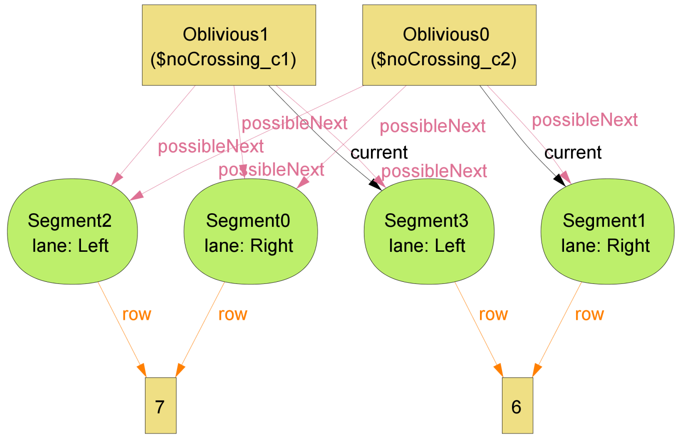
The `NormalAvoidLaneChange` policy is very similar to the `Paranoid` policy, but unlike the `Paranoid` policy, the `NormalAvoidLaneChange` policy passes the `noDeadlock` assertion. In the `Paranoid` policy `noDeadlock` counterexample shown in Figure 3.11, the adjacent cars were unable to move forward. Each car assumed that the other included the forward and diagonal segments in `possibleNext` while, actually, neither did. In `NormalAvoidLaneChange`, adjacent cars both exclude the diagonal segment from `possibleNext`. However, they both retain the forward segment in `possibleNext` and are able to make forward progress.

4.1.4 The `noCrossing` assertion

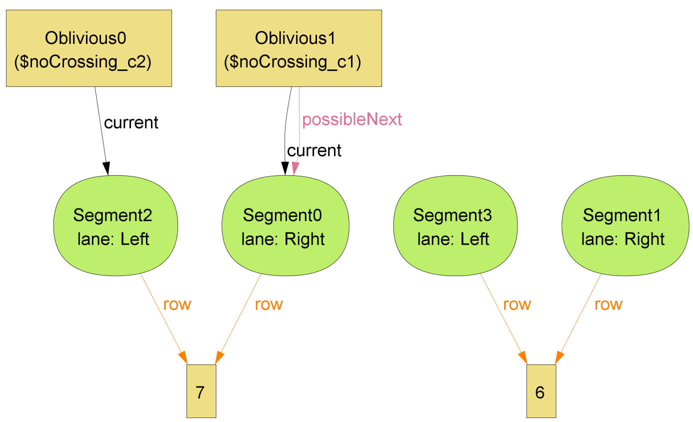
The `noCollision` property checks for collisions within a given time point. However, there is another type of collision can occur in the transition between time points. A *crossing collision* is the occurrence where two adjacent cars attempt to change lanes at the same time. The `noCollision` property cannot detect this type of collision, so we created the `noCrossing` property.

Definition 13 (The `noCrossing` property) *A crossing collision occurs when adjacent cars attempt to perform a lane-change maneuver at the same time. Two sequential time points,* *`pre` and `post` have the `noCrossing` property if there are no two cars `c1` and `c2` such that:*

- *Cars `c1` and `c2` are adjacent in the `pre` time point.*
- *Cars `c1` and `c2` are adjacent in the `post` time point.*
- *Car `c1`'s `current` segment in `post` is diagonal to `c1`'s `current` segment in `pre`*



(a) At Time0, Oblivious0 is in the right lane of row 6 and Oblivious1 is in the left lane of row 6.



(b) At Time1, Oblivious0 moves into the left lane of row 7 and Oblivious1 crosses into the right lane of row 7.

Figure 4.3: Example of a `noCrossing` collision between cars following the `Oblivious` policy

This different type of collision is an artifact of the level of abstraction chosen for the specification. We chose to define a car as occupying exactly one segment at every time point. This prevented the need to define which *sets* of segments a car could occupy simultaneously. For example, a car could occupy four segments at a time by having one wheel in each segment.

It also avoided edge cases where two cars occupy the same segment but may not collide, such as when one car is in the front of the segment and another is in the back.

This design choice does have tradeoffs. Here, we needed to create an additional specification to address crossing collisions that is more difficult to read than the specification of `noCollision`. If, in future work, we wanted to include motorcycles that safely drive in between lanes, we would encounter the same simultaneous occupancy complications mentioned in the previous paragraph.

The counterexample in Figure 4.3 shows cars following the `Oblivious` policy. The `NormalAvoid` policy passes `noCrossing` trivially because the policy does not allow lane-changing. The `NormalAvoidLaneChange` policy also passes the noCrossing assertion.

4.2 Development of Connected policies

Connected autonomous vehicles can gather more information via connected technology and use it to inform their driving decisions. This differs from human drivers that make decisions based on visually observable clues like the location of other cars.

A `Connected` policy describes the programmed driving behavior of a connected autonomous vehicle. Like the `Normal` policies, a `Connected` policy should first be safe on its own. Unlike the `Normal` policies, the `Connected` policy does not require unpredictability. Indeed, if a `Connected` policy can specify exactly one segment in `possibleNext`, ideally a forward or diagonal segment, its behavior will be predictable in other `Connected` policies. This may allow `Connected` policy-following vehicles to drive in tighter formation for improved road efficiency.

Connected vehicles must integrate with established traffic behaviors, and they bear the responsibility of ensuring safety in mixed traffic. When we found counterexamples to our property assertions in mixed traffic, we chose to modify the `Connected` policy rather than the `Normal` policy.

The `Connected` policy is allowed to be complicated; the only requirement is that the vehicle can decide a course of action quickly enough to react to other cars. In contrast, the Normal policy ideally does not behave differently in the presence of connected vehicles. In this work, we assume that the connected vehicle has unlimited computational resources and makes driving decisions instantaneously. Further work, possibly using complexity analysis, can determine if a particular policy is feasible for real-time driving.

The four policies described in this section are as follows: ConnectedI is a policy that allows cars to move forward or stop and broadcasts the segments in possibleNext to other connected vehicles. ConnectedII adds a filter that makes it safe around Normal vehicles. ConnectedIII incorporates lane-change functionality and is safe around Normal vehicles. ConnectedIV adds a strategy for picking the most productive segment to be possibleNext.

4.2.1 Driving Policy: ConnectedI

The main benefit of the connected vehicle is the additional information that can be gained from the connected technology. There are many different things connected cars might communicate with each other for better driving. In the Connected policies described in this work, connected cars choose to communicate their possible future locations, the segments in possibleNext, to other connected cars. Connected cars make use of this information by excluding segments "claimed" by other cars from their own set of possibleNext. This is a different behavior than the Paranoid policy that excluded segments that appeared to be within reach of other cars.

Definition 14 (The AvoidConnectedPossibleNextExceptSelf filter) *For a given Car and Time, the filter AvoidConnectedPossibleNextExceptSelf returns the current segment. The filter also returns all segments that are not occupied by cars following a Connected policy.*

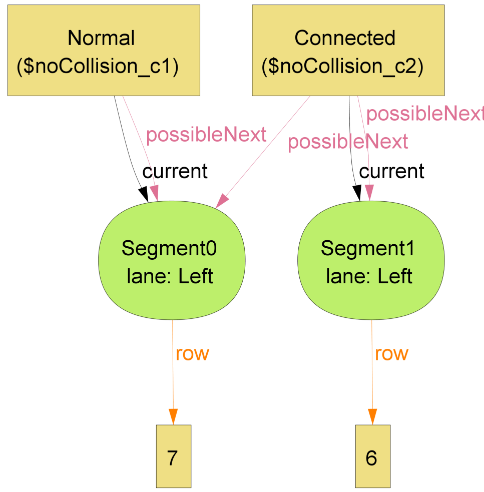
AvoidConnectedPossibleNextExceptSelf includes the current segment as a way of ensuring that the possibleNextNotEmpty assertion passes.

In real life, there is a negotiation between connected vehicles about which car has priority to occupy a space on the road. This negotiation might factor in if a car has access to other spaces, or if one car has higher priority than the other. Such a negotiation, if poorly managed, may take a significant amount of time to finish. Thus, it is important to check that a prospective negotiation protocol will always resolve conflict within a reasonable amount of time. (see Ploeg et al. for a discussion on timely autonomous decision making [11]).

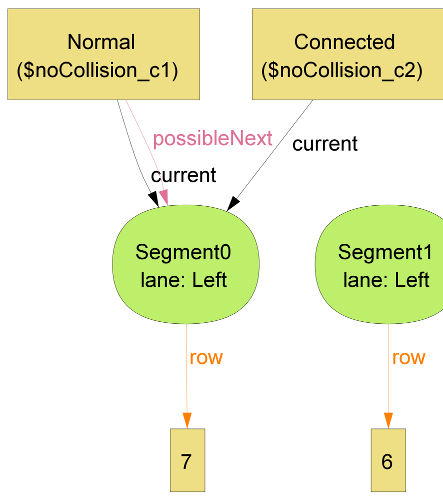
We did not design this Alloy specification to test negotiation protocols. At our specified level of abstraction, we assume that vehicles can communicate information instantaneously. The result of the `AvoidConnectedPossibleNextExceptSelf` filter is that no segment exists in two connected vehicle's `possibleNext`. When Alloy checks assertions of the specification, it only generates instances that obey this filter. This means that Alloy checks all of the instances in which cars eventually decide who keeps the segment in `possibleNext`, but Alloy cannot assess instances where the negotiation fails.

Definition 15 (The `ConnectedI` policy) *For a given `Car` and `Time`, the `ConnectedI` policy defines `possibleNext` as the segments that are returned by both the `ForeOrStop` and `AvoidConnectedPossibleNextExceptSelf` filters.*

The `ConnectedI` policy passes the `possibleNextNotEmpty`, `noCollision`, `noCrossing`, and `noDeadlock` property assertions in homogeneous traffic. However, the policy fails `noCollision` and `noCrossing` in mixed traffic with Normal Avoid cars.



(a) At Time0, the `Normal` vehicle is in Segment0 and the `Connected` vehicle is in Segment1.



(b) At Time1, the `Connected` vehicle moves forward and collides with the `Normal` vehicle.

Figure 4.4: Counterexample to `noCollision` assertion in mixed traffic with `NormalAvoid` and `ConnectedI`

Figure 4.4 shows a counterexample to the `noCollision` assertion. The `Connected` car moves forward and rear-ends the stopped `Normal` car. The same counterexample exists in mixed traffic with the `NormalAvoidLaneChange` policy.

One explanation of why the assertion fails is because the `Connected` car takes no action to avoid non-connected cars. The `Normal` policy includes the `AvoidOccupiedExceptSelf` filter which prevents the car from rear-ending a `Connected` car. However, since the `Connected` policy does not have this filter, the `Connected` car rear-ends the `Normal` car. In `ConnectedII`, we add the `AvoidOccupiedExceptSelf` filter from the `Normal` policy and the `noCollision` assertion passes in mixed traffic.

There are other ways to explain the failure. For example, one could argue that the `Normal` car failed to get out of the way of the `Connected` car. Our decision to blame the `Connected` policy is based on our belief that the `Connected` policy should integrate with existing driving norms. If we believed that `Connected` vehicles should have priority access to the road, perhaps because they are on a road reserved for platooning, we may have blamed the `Normal` policy instead. This hints at the value of Alloy as a validation tool; a method for checking that a specification meets the needs of a customer. We will discuss this further in Chapter 5.

4.2.2 Driving Policy: `ConnectedII`

As mentioned above, the `ConnectedII` policy passes `noCollision` and `noCrossing` in mixed traffic with the addition of the `AvoidOccupiedExceptSelf` filter.

Definition 16 (The `ConnectedII` policy) *For a given `Car` and `Time`, the `ConnectedII` policy defines `possibleNext` as the segments that are returned by the conjunction of the `ForeOrStop`, `AvoidConnectedPossibleNextExceptSelf`, and `AvoidOccupiedExceptSelf` filters.*

It is possible to replace the `AvoidOccupiedExceptSelf` filter in the `ConnectedII` policy with a new filter, `AvoidNormalOccupiedExceptSelf`, that only avoids segments occupied by `Normal` vehicles.

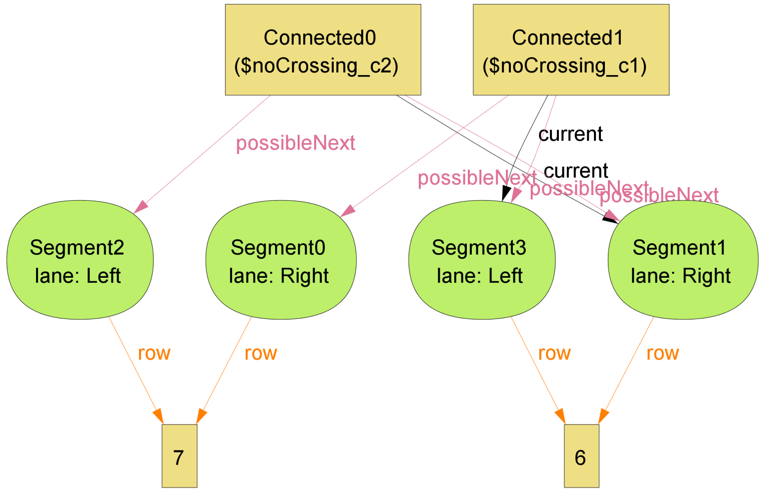
Definition 17 (The `AvoidNormalOccupiedExceptSelf` filter) *For a given `Car` and `Time`, the filter `AvoidNormalOccupiedExceptSelf` returns the `current` segment. The filter also returns all other segments that are not in a `Normal` car's `current`.*

The `AvoidConnectedPossibleNextExceptSelf` filter already prevents `Connected` cars from colliding with each other by virtue of the fact that the `current` segment is always in `possibleNext`. The proposed `AvoidNormalOccupiedExceptSelf` does not affect the results of analyzing mixed traffic, but it is technically correct. For simplicity, we use the `AvoidOccupiedExceptSelf` for `Connected` policies. The two filters are checked for equivalence in the appendix.

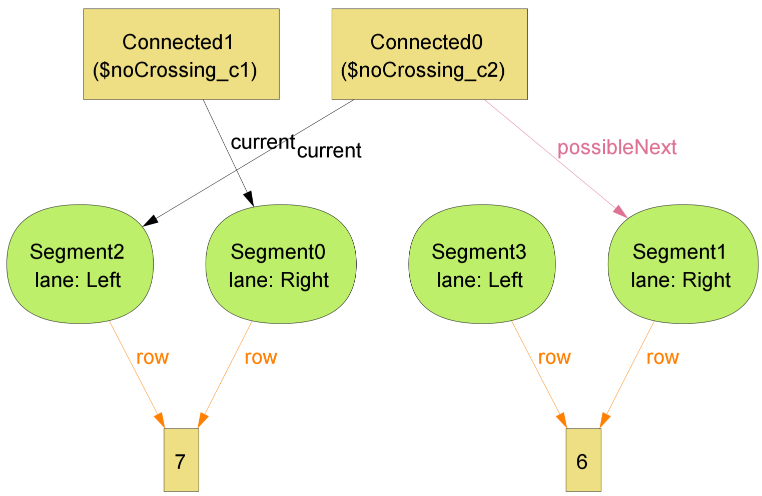
4.2.3 Driving Policy: `ConnectedIII`

The `ConnectedIII` policy is the `ConnectedII` policy with the addition of lane-changing capabilities.

If we exclude the `AvoidDiagonalIfAdjacentOccupied` filter that made NALC a safe policy, the scenario with all cars obeying `ConnectedIII` passes the `noCollision` assertion but fails the `noCrossing` assertion. In the scenario where all cars followed the `NormalAvoidLaneChange` policy, it was necessary to check for cars in the adjacent segment that might travel forward into the diagonal segment. In the `ConnectedIII` policy, an adjacent `Connected` car is able to inform the ego car if it intends to travel forward. If the adjacent car has the disputed segment in `possibleNext`, then the ego car will exclude it from `possibleNext`.



(a) At Time0, Connected0 is in the right lane of row 6 and Connected1 is in the left lane of row 6.



(b) At Time1, Connected0 moves diagonally into the left lane of row 7 while Connected1 moves diagonally into the right lane of row 7 resulting in a crossing collision.

Figure 4.5: Counterexample to `noCrossing` assertion in the `ConnectedIII` policy with the exclusion of the `AvoidDiagonalIfAdjacentOccupied` filter

However, the `ConnectedIII` policy without `AvoidDiagonalIfAdjacentOccupied` is susceptible to crossing collisions such as the one in Figure 4.5. Cars ensure no segments are shared across `possibleNext`, but the policy does not explicitly check whether a choice of `possibleNext` will inhibit a lane change maneuver.

When we include the `AvoidDiagonalIfAdjacentOccupied` filter, the `ConnectedIII` policy passes the `noCrossing` assertion. This is due to the fact that the `ConnectedIII` car, observing a car next to it, will not attempt a lane-change maneuver.

4.2.4 The `AvoidDiagonalIfNormalAdjacentElseCrossing` filter

Definition 18 (The `ConnectedIII` policy) *For a given `Car` and `Time`, the `ConnectedIII` policy defines `possibleNext` as the segments that are returned by the conjunction of the `ForeDiagOrStop`, `AvoidConnectedPossibleNextExceptSelf`, `AvoidOccupiedExceptSelf`, and `AvoidDiagonalIfAdjacentOccupied` filters.*

It is more technically correct, rather than reuse the `AvoidDiagonalIfAdjacentOccupied` filter from the `NormalAvoidLaneChange` policy, to invent a new filter that excludes the diagonal segment if either: (a) the adjacent car follows a `Normal` policy, or (b) the adjacent car follows a `Connected` policy and has a segment in `possibleNext` that may cause a collision (either the forward or the diagonal segment).

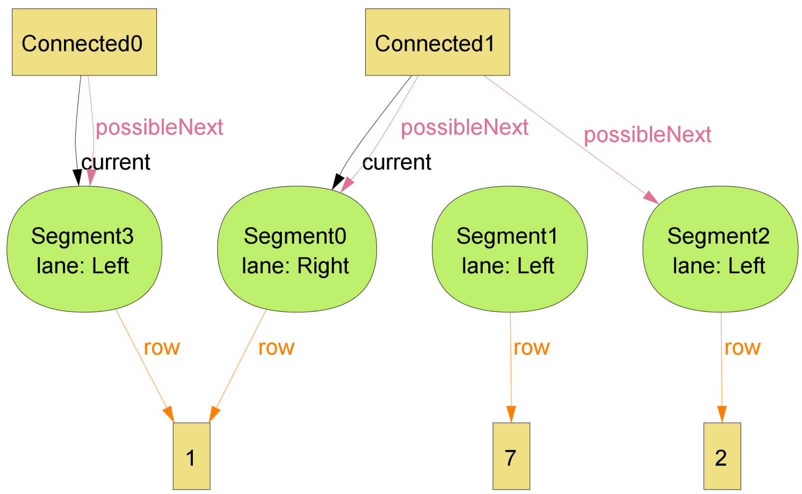
Definition 19 (The `AvoidDiagonalIfNormalAdjacentElseCrossing` filter) *For a given `Car` and `Time`, if the adjacent segment to the ego `Car` is occupied by a `Normal` car, the filter `AvoidDiagonalIfNormalAdjacentElseCrossing` excludes the diagonal segment. If the adjacent segment to the ego `Car` is occupied by a `Connected` car and the ego's fore segment is in the adjacent car's `possibleNext`, the filter `AvoidDiagonalIfNormalAdjacentElseCrossing` excludes the diagonal segment. The filter `AvoidDiagonalIfNormalAdjacentElseCrossing` returns all other segments.*

We hypothesized that this subtle nuance, like the `AvoidNormalOccupiedExceptSelf` filter in the `ConnectedII` policy, would have no effect. The two filters are compared for equivalence in Figure 4.6. The assertion states that, when `ConnectedIII` policy-following vehicles are in mixed traffic with `NormalAvoidLaneChange` vehicles, the alternative, technically-correct policy using the `AvoidDiagonalIfNormalAdjacentElseCrossing` filter behaves the same as the policy using the `AvoidDiagonalIfAdjacentOccupied` filter.

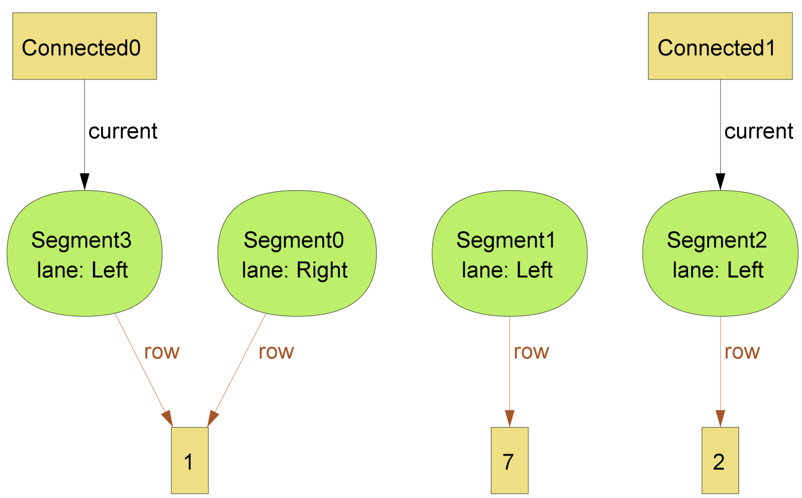
In this case, the technically correct version does make a functional difference. Figure 4.7 shows a scenario that the `AvoidDiagonalIfNormalAdjacentElseCrossing` filter allows that the other does not.

```
assert ACIIIbehavesLikeCIIInoCollision {
    all t: Time |
            MixedNALCConnectedIII[t]
            iff
            MixedNALCAlternativeConnectedIII[t]
}
check ACIIIbehavesLikeCIIInoCollision for 4 but 2 Car, 1 Time
// False
```

Figure 4.6: Comparing two policies for equivalency



(a) At Time0, Connected0 and Connected1 are in adjacent segments in row 1.



(b) At Time1, Connected1 safely crosses in front of Connected0 into the left lane of row 2.

Figure 4.7: Instance of a maneuver that `AvoidDiagonalIfNormalAdjacentElseCrossing` filter allows that `AvoidDiagonalIfAdjacentOccupied` filter does not

This result highlights the effects of over-specifying the policy. By using the `AvoidDiagonalIfAdjacentOccupied` filter from the `Normal` policy in the `Connected` policy, we failed to make full use of the information gained from communicating `possibleNext`. The `ConnectedIII` policy, with either the `AvoidDiagonalIfAdjacentOccupied` or `AvoidDiagonalIfNormalAdjacentElseCrossing`, is safe in mixed traffic. However, the `AvoidDiagonalIfNormalAdjacentElseCrossing` filter allows the connected vehicle to change lanes in front of another connected vehicle.

Definition 20 (The `ConnectedIII` policy) *For a given `Car` and `Time`, the `ConnectedIII` policy defines `possibleNext` as the segments that are returned by the conjunction of the `ForeDiagOrStop`, `AvoidConnectedPossibleNextExceptSelf`, `AvoidOccupiedExceptSelf`, and `AvoidDiagonalIfNormalAdjacentElseCrossing` filters.*

4.2.5 The `progress` assertion

So far, we have defined four properties for assessing the safety and productivity of our driving policies. The `possibleNextNotEmpty` property checked that the filter that composed a policy were not mutually exclusive. The `noCollision` and `noCrossing` properties checked that cars would not collide by occupying the same segment at the same time or crossing over each other. The `noDeadlock` property said that, as long as cars were not in gridlock, at least one car would have a segment in `possibleNext` other than its current segment.

All of the properties evaluated in this work are generally called *safety properties*, referring to the method in which they can be checked. Safety properties only require one time point (or two sequential time points) to find a counterexample that shows they are unsafe.

In contrast, *liveness properties* state that positive outcomes occur eventually. For example, liveness properties in autonomous vehicle design might state “all cars make it to their destinations eventually”, or “all cars make it to their destinations promptly.” Alpern and Schneider provide tests for determining whether a property is safety or liveness [47].

The productivity of driving protocols is actually a liveness property. Alloy is designed to test safety properties, but cannot reason about liveness properties. For this reason, we specified the productivity properties of `noDeadlock` and `progress` that can be understood with simple counterexamples.

Definition 21 (The `progress` property) *Two time points, `pre` and `post` have the `progress` property if there exists a car whose `current` segment in `pre` is different from its `current` segment in `post`.*

The `progress` property in says that in any two points in time, at least one car moves. The assertion of progress says that in any two sequential points of time, if cars follow the policy, then at least one car makes progress.

All of the `Normal` and `Connected` policies so far fail the progress assertion. This is, in part, because we chose to explicitly include the `current` segment in `possibleNext` to ensure the `possibleNextNotEmpty` property.

It is not significant that the `Normal` policies fail the `progress` assertion. The purpose of the `Normal` policy was to represent a baseline driving behavior that was safe and somewhat unpredictable. Human drivers are already motivated to reach their destinations, and if they have segments in `possibleNext` that get them closer to their destinations, they will choose to move there.

However, it is significant that the `Connected` policies fail the progress assertion. After ensuring safety, the `Connected` vehicle must attempt to reach its destination. All of our `Connected` policies so far fail the `progress` assertion, which means the vehicle may choose to remain stopped on the road indefinitely even if there is another segment closer to the destination.

4.2.6 Driving Policy: `ConnectedIV`

The `ConnectedIV` policy ensures `progress`. It accomplishes this by applying a strategy: when there are multiple segments that will not cause collisions, choose the segment that makes the most forward progress to be in `possibleNext`. This strategy assumes that the `Connected` car's destination is somewhere further down the road. Another potential destination could be any row in the right lane, perhaps because the Car is preparing to exit the road.

Definition 22 (The `ConnectedIV` policy) *For a given `Car` and `Time`, the `ConnectedIV` policy identifies safe segments as those segments that are returned by the conjunction of the `ForeDiagOrStop`, `AvoidConnectedPossibleNextExceptSelf`, `AvoidOccupiedExceptSelf`, and `AvoidDiagonalIfNormalAdjacentElseCrossing` filters. The `ConnectedIV` policy defines `possibleNext` as the first segment in the following ordered list to be present in the intersection of these filters: forward segment, diagonal segment, `current` segment.*

The `ConnectedIV` policy passes all five properties and is safe in mixed traffic with both `Normal` policies.

4.3 Summary of analysis

In this chapter, we developed several versions of `Normal` and `Connected` driving policies and analyzed them, both in homogeneous and mixed traffic, against five desireable properties of safety and productivity. The filters that compose the policies are summarized in Figure 4.8 and Figure 4.9.

The results of each property assertion for each policy in homogeneous traffic is summarized in Figure 4.10. As discussed in Chapter 3, the `Oblivious` policy fails the `noCollision` and `noCrossing` assertions, and the `Paranoid` policy fails the `noDeadlock` assertion. Only the `ConnectedIV` policy employs a strategy for picking the most productive `possibleNext` segment, and it is the only policy that passes the `progress` assertion.

Figure 4.11 summarizes the `noCollision` and `noCrossing` results of mixed traffic. The `noCollision` and `noCrossing` assertions both passed or failed, so the results in this table apply to both assertions. Only the `ConnectedI` policy was unsafe with `Normal` cars in mixed traffic.

Filter Name	Description
ForeOrStop	Returns all segments that are in front of or currently occupied by the ego car
ForeDiagOrStop	Returns all segments that are in front of, diagonal to, or currently occupied by the ego car
AvoidForeDiagOrStopOfPeer-ExceptSelf	Returns all segments that are **not** in front of, diagonal to, or currently occupied by another car, as well as the segment currently occupied by the ego car
AvoidOccupiedExceptSelf	Returns all segments that are **not** occupied by another car, as well as the segment currently occupied by the ego car
AvoidDiagonal-IfAdjacentOccupied	Returns all segments, except if there is a car next to the ego car, then it excludes the segment diagonal to the ego car (in front of the other car)
AvoidDiagonalIfNormal-Adjacent ElseCrossing	Returns all segments, except if there is a car adjacent to the ego car, then if the adjacent car is Normal, it excludes the segment in front of the adjacent car. Else if the adjacent car is Connected, it excludes the segment in front of the adjacent car if the adjacent car has forward or diagonal segment in PossibleNext
AvoidConnectedPossibleNext-ExceptSelf	Returns all segments that are not present in another `Connected` car's set of `possibleNext`, as well as the segment currently occupied by the ego car

Figure 4.8: Informal descriptions of the filters used in the `Oblivious`, `Paranoid`, `Normal`, and `Connected` driving policies

4.4 Modeling insights

The main goal of this work was to understand how Alloy can be used to make assertions about policies. We used the `Oblivious`, `Paranoid`, `Normal`, and `Connected` policies to show the ways that assertions might fail and how different kinds of assertions are needed to check the contrasting goals safety and productivity. These policies and properties were used to demonstrate the design choices and considerations when modeling with Alloy. The key results of our modeling are as follows:

- The abstraction of `possibleNext` as a set segments can model unpredictable human drivers by allowing multiple segments in `possibleNext`. Communicating the contents of `possibleNext` amongst `Connected` vehicles can model the connected capabilities of

Policy	**Filters on Possible Next**
Oblivious	ForeDiagOrStop
Paranoid	ForeDiagOrStop & AvoidForeDiagOrStopOfPeerExceptSelf
Normal Avoid	ForeOrStop & AvoidOccupiedExceptSelf
Normal Avoid with Lane Change	ForeDiagOrStop & AvoidOccupiedExceptSelf & AvoidDiagonalIfAdjacentOccupied
Connected I	ForeOrStop & AvoidOccupiedExceptSelf
Connected II	ForeOrStop & AvoidConnectedPossibleNextExceptSelf & AvoidOccupiedExceptSelf
Connected III	ForeDiagOrStop & AvoidConnectedPossibleNextExceptSelf & AvoidOccupiedExceptSelf & AvoidDiagonalIfNormalAdjacentElseCrossing
Connected IV	Most "productive" segment from: (ForeDiagOrStop & AvoidConnectedPossibleNextExceptSelf & AvoidOccupiedExceptSelf & AvoidDiagonalIfNormalAdjacentElseCrossing)

Figure 4.9: Summary of the `Oblivious`, `Paranoid`, `Normal`, and `Connected` driving policies according to their filters on `possibleNext`.

autonomous vehicles.

- The `NormalAvoidLaneChange` policy, unlike the `Paranoid` policy, was able to avoid deadlock scenarios by reasoning that an adjacent car would not attempt a lane-change maneuver.

- The `ConnectedI` policy showed that two policies that are safe on their need not be safe in mixed traffic.

- Alloy can show instances of maneuvers that are permitted under one policy but excluded from another. Figure 4.7 shows an example of a `Connected` car safely cutting in front of another `Connected` car with the additional knowledge that the other car is

Policy	**Assertions when all cars follow the same policy**				
	possibleNext-NotEmpty	noCollision	noCrossing	noDeadlock	progress
Oblivious	True	**False**	**False**	True	False
Paranoid	True	True	True	**False**	False
Normal Avoid	True	True	True	True	False
NALC	True	True	True	True	False
Connected I	True	True	True	True	False
Connected II	True	True	True	True	False
Connected III	True	True	True	True	False
Connected IV	True	True	True	True	**True**

Figure 4.10: Results of asserting of `possibleNextNotEmpty`, `noCollision`, `noCrossing`, `noDeadlock`, and `progress` properties on homogeneous traffic. Bold font identifies unusual results.

planning to stop.

- Human drivers are motivated to make progress, but connected policies need to be explicitly instructed. If a connected policy has multiple safe options, the policy needs to determine which one they should go to.

	Normal Avoid	NALC	Connected I	Connected II	Connected III	Connected IV
Normal Avoid	True	True	**False**	True	True	True
NALC		True	**False**	True	True	True
Connected I			True	True	True	True
Connected II				True	True	True
Connected III					True	True
Connected IV						True

Figure 4.11: Results of asserting `noCollision` and `noCrossing` in mixed traffic. Bold font identifies unusual results.

5 Discussion

This work was successful at checking safety and productivity properties of connected autonomous vehicle driving policies. The following are the key outcomes from this analysis.

Automated model checking allows us to reason about complicated policies.

Our first policies, `Oblivious` and `Paranoid`, had obvious flaws; the `Oblivious` policy did not protect against collisions, and the `Paranoid` policy cased deadlocks. The Alloy analyzer found counterexamples to our `noCollision` and `noDeadlock` assertions that helped us understand how our policies were flawed. As we developed `Normal` and `Connected` driving policies that were comprised of several filters, Alloy allowed us to automatically check the safety and productivity properties of our policies.

The level of abstraction determines what can be learned about a system.

By using the abstraction of cars in road segments, we defined cars as occupying exactly one segment at a time. This make it easy to define the `noCollision` property where cars occupy the same segment at the same time. However, with this level of granularity, we had to define the `noCrossing` property to check for collisions that occur between time points. The discrete representation of time worked well to reason about the safety of pre-determined driving decisions but prevented us from assessing the ability of driving protocols to swerve or accelerate to avoid potential collisions.

Safety, productivity, and efficiency are complementary goals.

As the `Paranoid` policy showed, safety alone allows for driving protocols that do not move. In contrast, the `Oblivious` policy showed that productivity without safety results in collisions.

These two goals helped us reason about the properties of driving protocols that make them useful.

A third goal, *efficiency*, is also necessary to ensure that a driving protocol is useful. A safe and productive yet inefficient protocol might cause cars to maintain a large gap between themselves and neighboring cars and would cause bottleneck congestion on busy roads. A safe and efficient yet unproductive protocol might result in cars assembling into a gridlock. Gridlocks are the optimal example of road use efficiency, and once cars are no longer able to proceed, they safely stop. Finally an efficient and productive yet unsafe protocol might attempt to construct platoons of vehicles moving in tight formation at high speeds. Without safety measures that account for the unpredictability of human drivers, these protocols might result in high-speed crashes.

5.1 Future work

As mentioned in Chapter 2, the automotive research community is interested in the application of connected technology for cooperative autonomous behaviors such as platooning.

Platooning is a natural extension of our models. Our driving policies mostly focused on choosing safe maneuvers, but the `ConnectedIV` policy introduced the notion of a driving strategy to pick the forward-most segment from the set of safe segments. Strategies can be used to enact perform higher-level platooning operations like assembling into a tight formation.

Platooning also introduces classical distributed system problems like concurrency and leader election. Zave demonstrated the efficacy of Alloy to uncover flaws in the Chord distributed hash table protocol [45]. Future work may seek to understand the impact of mixed driving on the operation of platoon management systems.

The Alloy analyzer can be used to assess *safety properties*, or properties that can be invalidated by the discovery of a counterexample. Recent efforts extend Alloy to reason about events happening over a many time points [48]. Cunha explores the use of the temporal logic in Alloy to assert liveness properties [49].

5.2 Conclusion

We used Alloy to develop and test autonomous vehicle driving protocols. We contributed two driving protocols that represent the autonomous and human-driven vehicles present in mixed traffic. Using five properties of safety and productivity, we showed our driving protocols were safe in mixed traffic. This thesis represents a case study in the ways that lightweight formal modeling can be used to reason about driving protocols early in the development process.

Appendices

Here is the Alloy specification for the work described in the thesis.

C Physical specification

```
/* =============================================================================
A DEFINITION OF THE PHYSICAL WORLD
MaryAnn VanValkenburg, Spring 2020

The universe is defined in terms of Cars, Segments, and Time. Cars occupy
segments. Cars can move between segments as a function of time.

Segments are discrete units of road. They have a lane and a row. The road is
currently constrained to two lanes, labeled left and right. The rows are
positive integers. Larger integers mean further ahead on the road.

A table (relation), called current (see sig Car), records the physical
position (Segment) of each car at each time.

A table (relation), called possibleNext (see Car), records the
segments a car may occupy in the next sequential unit of time. This is
generated by applying the car's driving policy to the current segment.

============================================================================= */

module physical
open util/ordering[Time] as trace

/* =============================================================================
TEMPORAL STRUCTURE
    + Event-based idiom
    + Time uses the util/ordering module
    + exactlyPrecedes predicate for convenience of comparing two units of time
============================================================================= */

sig Time { }

pred exactlyPrecedes [pre, post: Time] {
    post = pre.next
}

/* =============================================================================
PHYSICAL WORLD DEFINITIONS
    + Segments are physical units that have a unique fixed position
    + Segments are a lane (Left/Right) and a row (positive Int)
    + Cars exist in segments
    + Segments can be vacant or occupied by one or more cars
    + current.t.next is taken randomly from the set of possibleNext.t
============================================================================= */

/* ================================ SEGMENT ================================= */
// A segment has a lane and a row.
```

```
// Int uses util/ordering module.
// Larger Int row means further ahead on the road.
// Road stretches forward infinitely.

sig Segment {
    lane: one Lanes,
    row: one Int,
}

// Currently, just a two-lane road.
abstract sig Lanes {}
one sig Left extends Lanes {}
one sig Right extends Lanes {}

// Larger integer means further ahead on the road
// Simply because it is easier to reason about positive numbers
fact rowMustBePositive {
    all r: Segment.row | r ≥ 0
}

// Uniqueness of segments
fact sameRowSameLaneImpliesSameSegment {
    all s1, s2: Segment |
        s1.row = s2.row and s1.lane = s2.lane implies s1 = s2
}

/* ================================== CARS ================================== */

sig Car {
    current: Segment one -> Time, // Where the car currently is
    possibleNext: Segment -> Time, // Where the car can go next per policy
}

// This is how current.t.next is related to possibleNext.t
// A random pick of possible next
fact nextCurrentLocDerivedFromPossibleNext {
    all c: Car, t: Time |
        some t.next implies
            c.current.(t.next) in c.possibleNext.t
}


/* ============================ DERIVE SEGMENTS ============================ */

// Straight ahead and same lane is fore
fun fore (c: Car, t: Time) : set Segment {
    // the set of segments such that for each segment s...
    {s: Segment |
        // s is in the next row
        s.row = (c.current.t).(row.next) and
        // and in the same lane
        s.lane = (c.current.t).lane
    }
}

// Ahead and switching lanes is diag
fun diag (c: Car, t: Time) : set Segment {
    // the set of segments such that for each segment s...
    {s: Segment |
        // s is in the next row
        s.row = (c.current.t).(row.next) and
        // and in the other lane
        s.lane != (c.current.t).lane
    }
```

```
}

// Current segment is here
fun here (c: Car, t: Time) : set Segment {
    // the set of segments in which the car currently resides
    {c.current.t}
}

/* ============================ SANITY CHECKING ============================ */

// fore
pred foreIsAtLeastOne {
    all c: Car, t: Time |
        #fore[c, t] = 1
}
run foreIsAtLeastOne for 3

assert foreIsAtMostOne {
    all c: Car, t: Time |
        #fore[c, t] ≤ 1
}
check foreIsAtMostOne

// diag
pred diagIsAtLeastOne {
    all c: Car, t: Time |
        #diag[c, t] = 1
}
run diagIsAtLeastOne for 3

assert diagIsAtMostOne {
    all c: Car, t: Time |
        #diag[c, t] ≤ 1
}
check diagIsAtMostOne

// here
assert hereIsExactlyOneSegment {
    all c: Car, t: Time |
        c.possibleNext.t = here[c, t] implies
        // will always have this segment (because currently existing in!)
        #(c.possibleNext.t) = 1
}
check hereIsExactlyOneSegment for 5

// all
fun physicallyReachable (c: Car, t: Time) : set Segment {
    {s: Segment | s in fore[c, t] + diag[c, t] + here[c, t]}
}

pred physicallyReachableIsAtLeastThree [c: Car, t: Time] {
        #physicallyReachable[c, t] = 3
}
run physicallyReachableIsAtLeastThree for 6 but 1 Time

assert physicallyReachableIsAtMostThree {
    all c: Car, t: Time |
        #physicallyReachable[c, t] ≤ 3
}
check physicallyReachableIsAtMostThree for 7
```

D Safety properties

```
/* ============================================================================
PROPERTIES OF THE PHYSICAL WORLD
    MaryAnn VanValkenburg, Spring 2020

This builds off of the physical module which defines Cars, Segments, and
Time.

============================================================================ */

module properties
open physical

/* ======================== POSSIBLE NEXT NOT EMPTY ========================= */
// Used to check that policy rules are not mutually exclusive.
// All cars must have at least one segment in possibleNext. It may be the car's
// current segment.
// PASSING CONDITION: possibleNextNotEmpty

pred possibleNextNotEmpty [t: Time] {
    all c: Car | some c.possibleNext.t
}

/* ================================ COLLISION =============================== */
// Collision is when different cars occupy the same segment at the same time.
// NOTE: collision is NOT reflexive
// PASSING CONDITION: noCollision

pred collision [c1, c2: Car, t: Time] {
    // Different cars
    c1 != c2
    // Same segment at the same time
    c1.current.t = c2.current.t
}

pred noCollision [t: Time] {
    no c1, c2: Car | collision[c1, c2, t]
}

/* ================================ CROSSING ================================ */
// Crossing is a type of collision in which two cars try switching lanes over
// each other in sequential points in time.
// Crossing is NOT reflexive.
// PASSING CONDITION: noCrossing

pred crossing [c1, c2: Car, pre, post: Time] {
    // Different cars
    c1 != c2

    // c1 and c2 are adjacent (same row, different lane)
    c1.current.pre.row = c2.current.pre.row
    c1.current.pre.lane != c2.current.pre.lane

    // they swap lanes
    c1.current.pre.lane = c2.current.post.lane

    // now adjacent but different row than before
    c1.current.post.row = c2.current.post.row
    c1.current.post.lane != c2.current.post.lane
    c1.current.pre.row != c1.current.post.row
}
```

```

pred noCrossing [pre, post: Time] {
    no c1, c2: Car | crossing[c1, c2, pre, post]
}

/* ================================ DEADLOCK ================================ */
// Deadlock occurs when no cars have possibleNext other than current.
// PASSING CONDITION: noDeadlock

pred noDeadlock [t: Time] {
    // There exists a car that has a segment other than current in possibleNext
    some c: Car | some c.possibleNext.t - c.current.t
}

pred EmptyFore [c: Car, t: Time] {
    some s: Segment |
        s in fore[c, t] and
        no other: Car | s in other.current.t
}

pred someEmptyFore [t: Time] {
    some c: Car | EmptyFore[c, t]
}

pred EmptyForeOrDiag [c: Car, t: Time] {
    some s: Segment |
        s in (fore[c, t] + diag[c, t]) and
        no other: Car | s in other.current.t
}

pred someEmptyForeOrDiag [t: Time] {
    some c: Car | EmptyForeOrDiag[c, t]
}

/* ================================ PROGRESS ================================ */
// Progress occurs when at least one car moves between two time points.
// PASSING CONDITION: progress

pred progress [pre, post: Time] {
    some c: Car | c.current.pre != c.current.post
}

/* ============================ SANITY CHECKING ============================ */

// Collision
run collision for 5 but 1 Time
run noCollision for 5 but 1 Time

assert collisionIsNotReflexive {
    no c: Car, t: Time |
        collision[c, c, t]
}
check collisionIsNotReflexive for 2
// True

assert collisionIsSymmetric {
    all c1, c2: Car, t: Time |
        collision[c1, c2, t] implies collision[c2, c1, t]
}
check collisionIsSymmetric for 2
// True

assert collisionIsTransitive {
    all c1, c2, c3: Car, t: Time |
```

```
        (
            collision[c1, c2, t] and
            collision[c2, c3, t]
        ) implies collision[c1, c3, t]
}
check collisionIsTransitive for 4
// False. Fails when c1 = c3

assert noCollisionImpliesNoDoublyOccupied {
    all t: Time |
        noCollision[t] implies
            no c1, c2: Car | c1 != c2 and c1.current.t = c2.current.t
}
check noCollisionImpliesNoDoublyOccupied for 5 but 1 Time
// True

// Crossing
run crossing for 5 but 2 Time
run noCrossing for 5 but 2 Time

assert crossingIsNotReflexive {
    no c: Car, pre, post: Time | crossing[c, c, pre, post]
}
check crossingIsNotReflexive for 5

assert crossingIsSymmetric {
    all c1, c2: Car, t1, t2: Time |
        crossing[c1, c2, t1, t2] implies crossing[c2, c1, t1, t2]
}

// Progress
assert progressImpliesNoDeadlock {
    all pre, post: Time |
        exactlyPrecedes[pre, post] and
        // noDeadlock is a necessary condition for progress
        progress[pre, post] implies noDeadlock[pre]
}
check progressImpliesNoDeadlock for 5
```

E Driving policies

```
/* ==============================================================================
OBLIVIOUS, PARANOID, NORMAL, AND CONNECTED DRIVING POLICIES
    MaryAnn VanValkenburg, Spring 2020
============================================================================== */
module policies
open physical
open properties


/* ============================ Oblivious Policy ============================ */
// The oblivious policy says cars can go forward, diagonally, or stop. They do
// not take any action with respect to the position of other cars.

sig Oblivious extends Car {}

// FILTER
fun ForeDiagOrStop (c: Car, t: Time) : set Segment {
    // + := set union
    fore[c, t] + diag[c, t] + here[c, t]
}

pred ObliviousPolicy [c: Car, t: Time] {
    c.possibleNext.t = ForeDiagOrStop[c, t]
    c in Oblivious
}

/* ============================= Paranoid Policy ============================= */
// The paranoid policy says cars can go forward, diagonally, or stop, but they
// should not travel to a segment into which another car may go.

sig Paranoid extends Car {}

// FILTER
fun AvoidForeDiagOrStopOfPeerExceptSelf (c: Car, t: Time) : set Segment {
    // The set of segments where for each segment s, s is not physically
    // reachable by another car
    {s: Segment | all peer: Car-c |
        s in c.current.t or
        s not in ForeDiagOrStop[peer, t]}
}

pred ParanoidPolicy [c: Car, t: Time] {
    // & := set intersection
    c.possibleNext.t = ForeDiagOrStop[c, t] &
                       AvoidForeDiagOrStopOfPeerExceptSelf[c, t]
    c in Paranoid
}

/* =========================== Normal Avoid Policy =========================== */
// Normal Avoid cars can go forward or stop. If the fore segment is occupied,
// they cannot move forward.

sig Normal extends Car {} // Human-operated vehicle

// FILTER
fun ForeOrStop (c: Car, t: Time) : set Segment {
    fore[c, t] + here[c, t]
}

// FILTER
```

```
fun AvoidOccupiedExceptSelf (c: Car, t: Time) : set Segment {
    // Set of segments not occupied by other cars
    {s: Segment | s not in (Car-c).current.t}
}

pred NormalAvoidPolicy [c: Car, t: Time] {
    c.possibleNext.t = ForeOrStop[c, t] &
                       AvoidOccupiedExceptSelf[c, t]
    c in Normal
}

/* ==================== Normal Avoid Lane Change Policy ===================== */
// GOAL: Advance the normal driving policy to allow changing lanes
// Normal Avoid Policy with the additional rule: can change lanes as long as no
// one is beside you.


// Returns the set of cars that are beside the ego car
fun adjacent (c: Car, t: Time) : set Car {
    {peer: Car |
        // Same row
        c.current.t.row = peer.current.t.row and
        // Different lane
        c.current.t.lane != peer.current.t.lane}
}

// FILTER
fun AvoidDiagonalIfAdjacentOccupied (c: Car, t: Time) : set Segment {
    {s: Segment | all peer: adjacent[c, t] | s not in fore[peer, t] }
}

pred NormalAvoidLaneChangePolicy [c: Car, t: Time] {
    c.possibleNext.t =
        ForeDiagOrStop[c, t] &  // Can now go diagonally
        AvoidOccupiedExceptSelf[c, t] &
        AvoidDiagonalIfAdjacentOccupied[c, t]
    c in Normal
}

/* ============================ Connected I Policy ============================ */
// Connected cars can go forward or stop. Connected cars "broadcast" their
// possible next segments to other connected cars. No connected cars share
// possibleNext segments.

sig Connected extends Car {} // Connected autonomous vehicle

// FILTER
fun AvoidConnectedPossibleNextExceptSelf (c: Car, t: Time) : set Segment {
    // Set of segments not in possibleNext of other connected cars
    {s: Segment | s in c.current.t or s not in (Connected-c).possibleNext.t}
}

pred ConnectedIPolicy[c: Car, t: Time] {
    c.possibleNext.t = ForeOrStop[c, t] &
                       AvoidConnectedPossibleNextExceptSelf[c, t]
    c in Connected
}

/* ========================= Connected II Policy ========================= */
// Normal Avoid and Connected I did not prevent collision because Connected did
// not avoid currently occupied segments. Connected II amends Connected I to
// include AvoidOccupiedExceptSelf predicate, just like Normal Avoid.

pred ConnectedIIPolicy [c: Car, t: Time] {
```

```
    c.possibleNext.t = ForeOrStop[c, t] &
                       AvoidConnectedPossibleNextExceptSelf[c, t] &
                       AvoidOccupiedExceptSelf[c, t]
    c in Connected
}

/* =================== Connected III Policy =================== */
// Connected II but with lane change (and check adjacent rule)

// FILTER
fun AvoidDiagonalIfNormalAdjacentElseCrossing (c: Car, t: Time) : set Segment {
    {
        {s: Segment |
            // All normal peers
            all peer: adjacent[c, t] & Normal |
                // s is not in the peer's fore segment
                s not in fore[peer, t]
        }
    &
        {s: Segment |
            // All connected peers
            all peer: adjacent[c, t] & Connected |
                    // if fore[peer] in peer's possible next, exclude it
                    s not in (peer.possibleNext.t & fore[peer, t]) and
                    // if diag[peer] in peer's possible next, will have crossing
                    // collision, so exclude fore[peer]
                    s not in adjacent_segment[peer.possibleNext.t & diag[peer, t]]
         }
    }
}

fun adjacent_segment (s: Segment) : set Segment {
    {t: Segment | t.row = s.row and t.lane != s.lane}
}

pred ConnectedIIIPolicy [c: Car, t: Time] {
    c.possibleNext.t =
        ForeDiagOrStop[c, t] &
        AvoidConnectedPossibleNextExceptSelf[c, t] &
        AvoidOccupiedExceptSelf[c, t] &
        AvoidDiagonalIfNormalAdjacentElseCrossing[c, t]
    c in Connected
}

/* ========================= Connected IV Policy ========================= */
// Connected III policy with the additional strategy that the car will
// prioritize the fore segment, then the diag, then the stop segment in
// possibleNext.

fun ConnectedIIIPolicySegments (c: Car, t: Time) : set Segment {
        ForeDiagOrStop[c, t] &
        AvoidConnectedPossibleNextExceptSelf[c, t] &
        AvoidOccupiedExceptSelf[c, t] &
        AvoidDiagonalIfNormalAdjacentElseCrossing[c, t]
}

pred ConnectedIVPolicy [c: Car, t: Time] {
    c in Connected

    // if
    (some fore[c, t] & ConnectedIIIPolicySegments[c, t])
    // then
    implies (c.possibleNext.t = fore[c, t])
    // else
```

```
    else
    (
        // if
        (some diag[c, t] & ConnectedIIIPolicySegments[c, t])
        // then
        implies (c.possibleNext.t = diag[c, t])
        // else; should just be here[c, t]
        else (c.possibleNext.t = ConnectedIIIPolicySegments[c, t])
    )
}


/* ============================ SANITY CHECKING ============================= */

// ForeOrStop
run ForeOrStop for 6 but 1 Time

pred AllCarForeOrStop [t: Time] {
    all c: Car | c.possibleNext.t =  ForeOrStop[c, t]
}

assert allCarFOSImpliesNoCollision {
    all pre, post: Time |
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllCarForeOrStop[pre] implies
        noCollision[post]
}
check allCarFOSImpliesNoCollision for 5 but 2 Time
// Collision. Forward-moving car rear-ends a stopped car.

assert ForeOrStopIsTwo {
    all c: Car, t: Time |
        some fore[c, t] implies #ForeOrStop[c, t]=2
}
check ForeOrStopIsTwo for 5

assert ForeOrStopIsAlwaysAtLeastOne {
    all c: Car, t: Time | #ForeOrStop[c, t]≥1
}
check ForeOrStopIsAlwaysAtLeastOne for 5

assert ForeOrStopAlwaysIncludesCurrent {
    all c: Car, t: Time |
        c.current.t in ForeOrStop[c, t]
}
check ForeOrStopAlwaysIncludesCurrent for 5


// AvoidOccupiedExceptSelf
run AvoidOccupiedExceptSelf for 5 but 1 Time

assert SelfSegmentInAvoidOccupiedExceptSelf {
    all c: Car, t: Time | noCollision[t] implies
        c.current.t in AvoidOccupiedExceptSelf[c, t]
}
check SelfSegmentInAvoidOccupiedExceptSelf for 5 but 1 Time

pred AllCarAvoidOccupiedExceptSelf [t: Time] {
    all c: Car | c.possibleNext.t = AvoidOccupiedExceptSelf[c, t]
}

assert AllCarAvoidOccupiedExceptSelfNoCollision {
    all pre, post: Time |
```

```
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllCarAvoidOccupiedExceptSelf[pre] implies
        noCollision[post]
}
check AllCarAvoidOccupiedExceptSelfNoCollision for 5 but 2 Time
// Not safe on its own, two cars attempt to move to same vacant segment

// AvoidConnectedPossibleNextExceptSelf
run AvoidConnectedPossibleNextExceptSelf for 5 but 1 Time

pred AllCarConnectedAvoidConnectedPossibleNext [t: Time] {
    all c: Car |
        c in Connected and
        c.possibleNext.t = AvoidConnectedPossibleNextExceptSelf[c, t]
}
run AllCarConnectedAvoidConnectedPossibleNext for 5 but 1 Time

assert AllCarConnectedAvoidConnectedPossibleNextNoCollision {
    all pre, post: Time |
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllCarConnectedAvoidConnectedPossibleNext[pre] implies
        noCollision[post]
}
check AllCarConnectedAvoidConnectedPossibleNextNoCollision for 5
// Safe (when all cars are connected)

// Adjacent
assert adjacentNotReflexive {
    all c: Car, t: Time | c not in adjacent[c, t]
}
check adjacentNotReflexive for 5

assert noAdjacent {
    no c: Car, t: Time | some adjacent[c, t]
}
check noAdjacent for 5 but 1 Time
// want this to fail, meaning that adjacent is possible (shows examples of adjacent)

pred showSegmentsNotAdjacentFore [c: Car, t: Time] {
    c.possibleNext.t = physicallyReachable[c,t] &
                       AvoidDiagonalIfAdjacentOccupied[c, t]
    #Segment ≥ 6
}
run showSegmentsNotAdjacentFore for 7 but 1 Time
```

F Analysis of Oblivious and Paranoid driving policies

```
/* ==============================================================================
ANALYSIS OF OBLIVIOUS AND PARANOID DRIVING POLICIES
    MaryAnn VanValkenburg, Spring 2020
============================================================================== */
open properties
open policies


/* ======================== Scenario: All Oblivious ========================= */
pred AllOblivious [t: Time] {
    all c: Car | ObliviousPolicy[c, t]
}

pred showSafeOblivious [pre, post: Time] {
    exactlyPrecedes[pre, post]
    AllOblivious[pre]
    AllOblivious[post]
    noCollision[pre]
    noCollision[post]
    Car.current.pre != Car.current.post
    #Car = 2
}
run showSafeOblivious for 5 but 2 Time

assert AOpossibleNextNotEmpty {
    all t: Time |
    (
        AllOblivious[t]
    )
    implies possibleNextNotEmpty[t]
}
check AOpossibleNextNotEmpty for 5
// True

assert AOnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllOblivious[pre]
    )
    implies noCollision[post]
}
check AOnoCollision for 5 but 2 Time
// False. Rear-end a stopped car

assert AOnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllOblivious[pre]
    )
    implies noCrossing[pre, post]
}
check AOnoCrossing for 4 but 2 Car, 2 Time
// False. Adjacent cars swap lanes

assert AOnoDeadlock {
    all t: Time |
```

```
    (
        noCollision[t] and
        AllOblivious[t] and
        someEmptyForeOrDiag[t]
    )
    implies noDeadlock[t]
}
check AOnoDeadlock for 5 but 1 Time
// True

assert AOprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllOblivious[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check AOprogress for 5 but 2 Time
// False. No incentive to progress

/* ========================= Scenario: All Paranoid ========================= */
pred AllParanoid [t: Time] {
    all c: Car | ParanoidPolicy[c, t]
}

pred showAllParanoid [pre, post: Time] {
    exactlyPrecedes[pre, post]
    AllParanoid[pre]
    AllParanoid[post]
    noCollision[pre]
    noCollision[post]
    Car.current.pre != Car.current.post
    #Car = 2
}
run showAllParanoid for 5 but 2 Time

assert APpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllParanoid[t]
    )
    implies possibleNextNotEmpty[t]
}
check APpossibleNextNotEmpty for 5 but 1 Time
// True with addition of (+ c.current.t in the policy definition)

assert APnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllParanoid[pre]
    )
    implies noCollision[post]
}
check APnoCollision for 5 but 2 Time
// True

pred APrunning [pre, post: Time] {
        exactlyPrecedes[pre, post]
        noCollision[pre]
```

```
        AllParanoid[pre]
        AllParanoid[post]
        !noDeadlock[pre]
        #Car = 2
        all c: Car | EmptyForeOrDiag[c, pre]
}
run APrunning for 5 but 2 Time

assert APnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllParanoid[pre]
    )
    implies noCrossing[pre, post]
}
check APnoCrossing for 4 but 2 Car, 2 Time
// True

assert APnoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllParanoid[t] and
        someEmptyForeOrDiag[t]
    )
    implies noDeadlock[t]
}
check APnoDeadlock for 5 but 1 Time
// False. Two adjacent cars cancel each other out

assert APprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllParanoid[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check APprogress for 5 but 2 Time
// False. No incentive to progress

/* ================= Scenario: Mixed Oblivious or Paranoid ================== */
pred MixedObliviousOrParanoid [t: Time] {
    all c: Car | ObliviousPolicy[c, t] or ParanoidPolicy[c, t]
}

assert MOPnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedObliviousOrParanoid[pre]
    )
    implies noCollision[post]
}
check MOPnoCollision for 5 but 2 Time
// False. Oblivious car rear-ends Paranoid car

assert MOPnoCrossing {
    all pre, post: Time |
    (
```

```
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedObliviousOrParanoid[pre]
    )
    implies noCrossing[pre, post]
}
check MOPnoCrossing for 4 but 2 Car, 2 Time
// False. Inherits flaw from Oblivious Policy
```

G Analysis of Normal and Connected driving policies

```
/* ============================================================================
ANALYSIS OF NORMAL AND CONNECTED DRIVING POLICIES
    MaryAnn VanValkenburg, Spring 2020
============================================================================ */
open properties
open policies

/* ====================== Scenario: All Normal Avoid ====================== */
pred AllNormalAvoid [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t]
}

pred showAllNormalAvoid [pre, post: Time] {
    exactlyPrecedes[pre, post]
    AllNormalAvoid[pre]
    AllNormalAvoid[post]
    noCollision[pre]
    noCollision[post]
    Car.current.pre != Car.current.post
    #Car = 2
}
run showAllNormalAvoid for 5 but 2 Time

assert ANApossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllNormalAvoid[t]
    )
    implies possibleNextNotEmpty[t]
}
check ANApossibleNextNotEmpty for 5 but 1 Time
// True

assert ANAnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNormalAvoid[pre]
    )
    implies noCollision[post]
}
check ANAnoCollision for 5 but 2 Time
// True

assert ANAnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNormalAvoid[pre]
    )
    implies noCrossing[pre, post]
}
check ANAnoCrossing for 4 but 2 Car, 2 Time
// True

assert ANAnoDeadlock {
    all t: Time |
```

```
    (
        noCollision[t] and
        AllNormalAvoid[t] and
        someEmptyFore[t] // Diag doesn't apply to this policy
    )
    implies noDeadlock[t]
}
check ANAnoDeadlock for 7
// True

assert ANAprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllNormalAvoid[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ANAprogress for 5 but 2 Time
// False. No incentive to progress

/* =============== Scenario: Naiive Normal Avoid Lane Change ================ */
// Naiive version without additional rule about checking for adjacent car
pred NormalAvoidLaneChangePolicyNaiive [c: Car, t: Time] {
    c.possibleNext.t =
        ForeDiagOrStop[c, t] &  // Can now go diagonally
        AvoidOccupiedExceptSelf[c, t]
    c in Normal
}

pred AllNALCNaiive [t: Time] {
    all c: Car | NormalAvoidLaneChangePolicyNaiive[c, t]
}

assert ANALCNnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNALCNaiive[pre]
    )
    implies noCollision[post]
}
check ANALCNnoCollision for 5 but 2 Time
// False

assert ANALCNnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNALCNaiive[pre]
    )
    implies noCrossing[pre, post]
}
check ANALCNnoCrossing for 5 but 2 Time
// False

/* ================ Scenario: All Normal Avoid Lane Change ================ */
pred AllNormalAvoidLaneChange [t: Time] {
    all c: Car | NormalAvoidLaneChangePolicy[c, t]
}

```

```
assert ANALCpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllNormalAvoidLaneChange[t]
    )
    implies possibleNextNotEmpty[t]
}
check ANALCpossibleNextNotEmpty for 5 but 1 Time
// True

assert ANALCnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNormalAvoidLaneChange[pre]
    )
    implies noCollision[post]
}
check ANALCnoCollision for 5 but 2 Time
// True

assert ANALCnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllNormalAvoidLaneChange[pre]
    )
    implies noCrossing[pre, post]
}
check ANALCnoCrossing for 4 but 2 Car, 2 Time
// True

assert ANALCnoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllNormalAvoidLaneChange[t] and
        someEmptyForeOrDiag[t] // Diag DOES help this policy
    )
    implies noDeadlock[t]
}
check ANALCnoDeadlock for 7
// True

assert ANALCprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllNormalAvoidLaneChange[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ANALCprogress for 5 but 2 Time
// False. No incentive to progress

/* ======================== Scenario: Mixed Normal ======================== */
pred MixedNormal [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t] or NormalAvoidLaneChangePolicy[c, t]
}
```

```
assert MNnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormal[pre]
    )
    implies noCollision[post]
}
check MNnoCollision for 5 but 2 Time
// True

assert MNnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormal[pre]
    )
    implies noCrossing[pre, post]
}
check MNnoCrossing for 4 but 2 Car, 2 Time
// True



/* =========================== Connected Policies =========================== */

/* ======================== Scenario: All Connected I ======================== */
pred AllConnectedI [t: Time] {
    all c: Car | ConnectedIPolicy[c, t]
}

assert ACIpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedI[t]
    )
    implies possibleNextNotEmpty[t]
}
check ACIpossibleNextNotEmpty for 5 but 1 Time
// True

assert ACInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedI[pre]
    )
    implies noCollision[post]
}
check ACInoCollision for 5 but 2 Time
// True

assert ACInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedI[pre]
    )
    implies noCrossing[pre, post]
```

```
}
check ACInoCrossing for 4 but 2 Car, 2 Time
// True

assert ACInoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedI[t] and
        someEmptyFore[t] // Diag doesn't apply to this policy
    )
    implies noDeadlock[t]
}
check ACInoDeadlock for 5 but 1 Time
// True

assert ACIprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllConnectedI[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ACIprogress for 5 but 2 Time
// False. No incentive to progress

/* ============== Scenario: Mixed Normal Avoid or Connected I =============== */
pred MixedNormalAvoidConnectedI [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t] or ConnectedIPolicy[c, t]
}

assert MNACInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedI[pre]
    )
    implies noCollision[post]
}
check MNACInoCollision for 5 but 2 Time
// False. Connected car does not avoid segments occupied by Normal cars

assert MNACInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedI[pre]
    )
    implies noCrossing[pre, post]
}
check MNACInoCrossing for 4 but 2 Car, 2 Time
// True

/* ================== Scenario: Mixed NALC or Connected I ================== */
pred MixedNALCConnectedI [t: Time] {
    all c: Car | NormalAvoidLaneChangePolicy[c, t] or ConnectedIPolicy[c, t]
}

assert MNALCCInoCollision {
    all pre, post: Time |
```

```
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedI[pre]
    )
    implies noCollision[post]
}
check MNALCCInoCollision for 5 but 2 Time
// False. Connected car does not avoid segments occupied by Normal cars

assert MNALCCInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedI[pre]
    )
    implies noCrossing[pre, post]
}
check MNALCCInoCrossing for 4 but 2 Car, 2 Time
// True


/* ======================= Scenario: All Connected II ======================= */
pred AllConnectedII [t: Time] {
    all c: Car | ConnectedIIPolicy[c, t]
}

assert ACIIpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedII[t]
    )
    implies possibleNextNotEmpty[t]
}
check ACIIpossibleNextNotEmpty for 5 but 1 Time
// True

assert ACIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedII[pre]
    )
    implies noCollision[post]
}
check ACIInoCollision for 5 but 2 Time
// True

assert ACIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedII[pre]
    )
    implies noCrossing[pre, post]
}
check ACIInoCrossing for 4 but 2 Car, 2 Time
// True

assert ACIInoDeadlock {
```

```
    all t: Time |
    (
        noCollision[t] and
        AllConnectedII[t] and
        someEmptyFore[t] // Diag doesn't apply to this policy
    )
    implies noDeadlock[t]
}
check ACIInoDeadlock for 5 but 1 Time
// True

assert ACIIprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllConnectedII[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ACIIprogress for 5 but 2 Time
// False. No incentive to progress

/* ============== Scenario: Mixed Normal Avoid or Connected II ============== */
pred MixedNormalAvoidConnectedII [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t] or ConnectedIIPolicy[c, t]
}

assert MNACIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedII[pre]
    )
    implies noCollision[post]
}
check MNACIInoCollision for 5 but 2 Time
// True

assert MNACIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedII[pre]
    )
    implies noCrossing[pre, post]
}
check MNACIInoCrossing for 4 but 2 Car, 2 Time
// True

// Accomplished goal: Normal and Connected safely on the road together


/* ================== Scenario: Mixed NALC or Connected II ================== */
pred MixedNALCConnectedII [t: Time] {
    all c: Car | NormalAvoidLaneChangePolicy[c, t] or ConnectedIIPolicy[c, t]
}

assert NALCCIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
```

```
        noCollision[pre] and
        MixedNALCConnectedII[pre]
    )
    implies noCollision[post]
}
check NALCCIInoCollision for 5 but 2 Time
// True

assert NALCCIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedII[pre]
    )
    implies noCrossing[pre, post]
}
check NALCCIInoCrossing for 4 but 2 Car, 2 Time
// True

/* ============ Scenario: Mixed NALC or Alternative Connected II ============ */
// What if Connected II used the AvoidOccupiedExceptSelf rule instead of
// AvoidNormalOccupiedExceptSelf? Would they behave the same?

// RULE
fun AvoidNormalOccupiedExceptSelf (c: Car, t: Time) : set Segment {
    // Set of segments not occupied by Normal cars
    {s: Segment | s not in (Normal-c).current.t}
}

pred AlternativeConnectedIIPolicy [c: Car, t: Time] {
    c.possibleNext.t = ForeOrStop[c, t] &
                       AvoidConnectedPossibleNextExceptSelf[c, t] &
                       AvoidNormalOccupiedExceptSelf[c, t]
    c in Connected
}

pred MixedNALCAlternativeConnectedII [t: Time] {
    all c: Car |
        NormalAvoidLaneChangePolicy[c, t] or
        AlternativeConnectedIIPolicy[c, t]
}

assert ACIIbehavesLikeCII {
    all t: Time |
        (MixedNALCConnectedII[t] iff MixedNALCAlternativeConnectedII[t])
}
check ACIIbehavesLikeCII for 4 but 2 Car, 1 Time
// False. It appears that the counterexample is when two connected cars start in
// the same segment

assert ACIIbehavesLikeCIInoCollision {
    all t: Time |
        (
            noCollision[t] and
            MixedNALCConnectedII[t]
        )
            iff
        (
            noCollision[t] and
            MixedNALCAlternativeConnectedII[t]
        )
}
check ACIIbehavesLikeCIInoCollision for 4 but 2 Car, 1 Time
```

```
// True

assert ACIIbehavesLikeCIIProgress {
    all pre, post: Time |
        // MixedNALCConnectedII results in progress...
        ( (
            noCollision[pre] and
            MixedNALCConnectedII[pre]
          ) implies progress[pre, post]
        )
        iff
        // ... iff MixedNALCAlternativeConnectedII also results in progress
        ( (
            noCollision[pre] and
            MixedNALCAlternativeConnectedII[pre]
          ) implies progress[pre, post]
        )
}
check ACIIbehavesLikeCIIProgress for 4 but 2 Car, 2 Time
// True

/* ===================== Scenario: Naiive Connected III ===================== */
// Naiive version without additional rule about checking for adjacent car

pred ConnectedIIIPolicyNaiive [c: Car, t: Time] {
    c.possibleNext.t =
        ForeDiagOrStop[c, t] &
        AvoidConnectedPossibleNextExceptSelf[c, t] &
        AvoidOccupiedExceptSelf[c, t]
    c in Connected
}

pred AllConnectedIIINaiive [t: Time] {
    all c: Car | ConnectedIIIPolicyNaiive[c, t]
}

assert ACIIINnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedIIINaiive[pre]
    )
    implies noCollision[post]
}
check ACIIINnoCollision for 7

assert ACIIINnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedIIINaiive[pre]
    )
    implies noCrossing[pre, post]
}
check ACIIINnoCrossing for 4 but 2 Car, 2 Time


/* ====================== Scenario: All Connected III ====================== */
pred AllConnectedIII [t: Time] {
    all c: Car | ConnectedIIIPolicy[c, t]
}

```

```
assert ACIIIpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedIII[t]
    )
    implies possibleNextNotEmpty[t]
}
check ACIIIpossibleNextNotEmpty for 5 but 1 Time
// True

assert ACIIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedIII[pre]
    )
    implies noCollision[post]
}
check ACIIInoCollision for 5 but 2 Time
// True

assert ACIIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedIII[pre]
    )
    implies noCrossing[pre, post]
}
check ACIIInoCrossing for 4 but 2 Car, 2 Time
// True

assert ACIIInoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedIII[t] and
        someEmptyForeOrDiag[t]
    )
    implies noDeadlock[t]
}
check ACIIInoDeadlock for 5 but 1 Time
// True

assert ACIIIprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllConnectedIII[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ACIIIprogress for 5 but 2 Time
// False. No incentive to progress

/* ============ Scenario: Mixed Normal Avoid or Connected III ============= */
pred MixedNormalAvoidConnectedIII [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t] or ConnectedIIIPolicy[c, t]
}

```

```
assert MNACIIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedIII[pre]
    )
    implies noCollision[post]
}
check MNACIIInoCollision for 5 but 2 Time
// True

assert MNACIIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedIII[pre]
    )
    implies noCrossing[pre, post]
}
check MNACIIInoCrossing for 4 but 2 Car, 2 Time
// True

/* =========== Scenario: Mixed NALC or Alternative Connected III ============ */
// What if Connected III used AvoidDiagonalIfNormalAdjacentElseCrossing instead
// of the AvoidDiagonalIfAdjacentOccupied? Would it behave the same?

pred AlternativeConnectedIIIPolicy [c: Car, t: Time] {
    c.possibleNext.t =
        ForeDiagOrStop[c, t] &
        AvoidConnectedPossibleNextExceptSelf[c, t] &
        AvoidNormalOccupiedExceptSelf[c, t] &
        AvoidDiagonalIfAdjacentOccupied[c, t]
    c in Connected
}

pred MixedNALCAlternativeConnectedIII [t: Time] {
    all c: Car |
        NormalAvoidLaneChangePolicy[c, t] or
        AlternativeConnectedIIIPolicy[c, t]
}

assert ACIIIbehavesLikeCIIInoCollision {
    all t: Time |
        (
            // The only time they behave differently is crossing
            MixedNALCConnectedIII[t]
            iff
            MixedNALCAlternativeConnectedIII[t]
        ) or
        !noCollision[t]
}
check ACIIIbehavesLikeCIIInoCollision for 4 but 2 Car, 1 Time
// False

assert ACIIIbehavesLikeCIIIProgress {
    all pre, post: Time |
        // ... iff MixedNALCAlternativeConnectedIII also results in progress
        ( (
            exactlyPrecedes[pre, post] and
            MixedNALCAlternativeConnectedIII[pre]
          ) implies progress[pre, post]
        )
```

```
        iff
        // MixedNALCConnectedIII results in progress...
        ( (
            exactlyPrecedes[pre, post] and
            MixedNALCConnectedIII[pre]
          ) implies progress[pre, post]
        )
}
check ACIIIbehavesLikeCIIIProgress for 4 but 2 Car, 2 Time
// False!

/* ================= Scenario: Mixed NALC or Connected III ================== */
pred MixedNALCConnectedIII [t: Time] {
    all c: Car |
        NormalAvoidLaneChangePolicy[c, t] or
        ConnectedIIIPolicy[c, t]
}

assert MNALCCIIInoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedIII[pre]
    )
    implies noCollision[post]
}
check MNALCCIIInoCollision for 5 but 2 Time
// True

assert MNALCCIIInoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedIII[pre]
    )
    implies noCrossing[pre, post]
}
check MNALCCIIInoCrossing for 4 but 2 Car, 2 Time
// True


/* ======================= Scenario: All Connected IV ======================= */
pred AllConnectedIV [t: Time] {
    all c: Car | ConnectedIVPolicy[c, t]
}

assert ACIVpossibleNextNotEmpty {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedIV[t]
    )
    implies possibleNextNotEmpty[t]
}
check ACIVpossibleNextNotEmpty for 5 but 1 Time
// True

assert ACIVnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
```

```
        AllConnectedIV[pre]
    )
    implies noCollision[post]
}
check ACIVnoCollision for 5 but 2 Time
// True

assert ACIVnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        AllConnectedIV[pre]
    )
    implies noCrossing[pre, post]
}
check ACIVnoCrossing for 4 but 2 Car, 2 Time
// True

assert ACIVnoDeadlock {
    all t: Time |
    (
        noCollision[t] and
        AllConnectedIV[t] and
        someEmptyForeOrDiag[t]
    )
    implies noDeadlock[t]
}
check ACIVnoDeadlock for 5 but 1 Time
// True

assert ACIVprogress {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        AllConnectedIV[pre] and
        noDeadlock[pre]
    )
    implies progress[pre, post]
}
check ACIVprogress for 5 but 2 Time
// True

/* ======================= Scenario: Mixed Connected ======================= */
pred MixedConnected [t: Time] {
    all c: Car |
        ConnectedIPolicy[c, t] or
        ConnectedIIPolicy[c, t] or
        ConnectedIIIPolicy[c, t] or
        ConnectedIVPolicy[c, t]
}

assert MCnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedConnected[pre]
    )
    implies noCollision[post]
}
check MCnoCollision for 5 but 2 Time
// True

```

```
assert MCnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedConnected[pre]
    )
    implies noCrossing[pre, post]
}
check MCnoCrossing for 4 but 2 Car, 2 Time
// True

/* ============== Scenario: Mixed Normal Avoid or Connected IV ============== */
pred MixedNormalAvoidConnectedIV [t: Time] {
    all c: Car | NormalAvoidPolicy[c, t] or ConnectedIVPolicy[c, t]
}

assert MNACIVnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedIV[pre]
    )
    implies noCollision[post]
}
check MNACIVnoCollision for 5 but 2 Time
// True

assert MNACIVnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNormalAvoidConnectedIV[pre]
    )
    implies noCrossing[pre, post]
}
check MNACIVnoCrossing for 4 but 2 Car, 2 Time
// True

/* ================== Scenario: Mixed NALC or Connected IV ================== */
pred MixedNALCConnectedIV [t: Time] {
    all c: Car | NormalAvoidLaneChangePolicy[c, t] or ConnectedIVPolicy[c, t]
}

assert MNALCCIVnoCollision {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedIV[pre]
    )
    implies noCollision[post]
}
check MNALCCIVnoCollision for 5 but 2 Time
// True

assert MNALCCIVnoCrossing {
    all pre, post: Time |
    (
        exactlyPrecedes[pre, post] and
        noCollision[pre] and
        MixedNALCConnectedIV[pre]
```

```
    )
    implies noCrossing[pre, post]
}
check MNALCCIVnoCrossing for 4 but 2 Car, 2 Time
// True
```

Bibliography

[1] H. Oh, C. Yae, D. Ahn, and H. Cho, "5.8 GHz DSRC packet communication system for ITS services," in *Gateway to 21st Century Communications Village. VTC 1999-Fall. IEEE VTS 50th Vehicular Technology Conference (Cat. No. 99CH36324)*, vol. 4, pp. 2223–2227, IEEE, 1999.

[2] R. Miucic, A. Sheikh, Z. Medenica, and R. Kunde, "V2X applications using collaborative perception," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, pp. 1–6, IEEE, 2018.

[3] S. Moridpour, M. Sarvi, and G. Rose, "Lane changing models: A critical review," *Transportation letters*, vol. 2, no. 3, pp. 157–173, 2010.

[4] M. Atagoziyev, K. W. Schmidt, and E. G. Schmidt, "Lane change scheduling for autonomous vehicles," *IFAC-PapersOnLine*, vol. 49, no. 3, pp. 61–66, 2016.

[5] Y. Luo, G. Yang, M. Xu, Z. Qin, and K. Li, "Cooperative lane-change maneuver for multiple automated vehicles on a highway," *Automotive Innovation*, pp. 1–12, 2019.

[6] J. Erdmann, "Lane-changing model in SUMO," *Proceedings of the SUMO2014 modeling mobility with open data*, vol. 24, pp. 77–88, 2014.

[7] Y. Zhou, H. Zhu, M. Guo, and J. Zhou, "Impact of CACC vehicles' cooperative driving strategy on mixed four-lane highway traffic flow," *Physica A: Statistical Mechanics and its Applications*, p. 122721, 2019.

[8] U. P. Mudalige, "Platoon Vehicle Management," United States Patent 8,352,111 B2, Jan. 8, 2013.

[9] J. Kuhr, N. R. Juri, C. R. Bhat, J. Archer, J. C. Duthie, E. Varela, M. Zalawadia, T. Bamonte, A. Mirzaei, H. Zheng, *et al.*, "Travel modeling in an era of connected and automated transportation systems: An investigation in the Dallas-Fort Worth area.," tech. rep., University of Texas at Austin. Data-Supported Transportation Operations . . . , 2017.

[10] S. Eilers, J. Mårtensson, H. Pettersson, M. Pillado, D. Gallegos, M. Tobar, K. H. Johansson, X. Ma, T. Friedrichs, S. S. Borojeni, and M. Adolfson, "COMPANION – towards co-operative platoon management of heavy-duty vehicles," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pp. 1267–1273, Sept. 2015.

[11] J. Ploeg and R. de Haan, "Cooperative automated driving: From platooning to maneuvering," *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems*, 2019.

[12] M. Amoozadeh, *Towards Robust and Secure Collaborative Driving and Interactive Traffic Intersections.* University of California, Davis, 2018.

[13] H. Schweppe and Y. Roudier, "Security and privacy for in-vehicle networks," in *2012 IEEE 1st International Workshop on Vehicular Communications, Sensing, and Computing (VCSC)*, (Seoul, Korea (South)), pp. 12–17, IEEE, June 2012.

[14] M. Khajeh Hosseini, A. Talebpour, and S. Shakkottai, "Privacy risk of connected vehicles in relation to vehicle tracking when transmitting basic safety message type 1 data," *Transportation Research Record*, p. 0361198119875433, 2019.

[15] Y. Sun, L. Wu, S. Wu, S. Li, T. Zhang, L. Zhang, J. Xu, and Y. Xiong, "Security and Privacy in the Internet of Vehicles," in *2015 International Conference on Identification, Information, and Knowledge in the Internet of Things (IIKI)*, pp. 116–121, IEEE, 2015.

[16] B. K. Chaurasia, S. Verma, and G. Tomar, "Attacks on anonymity in VANET," in *2011 International Conference on Computational Intelligence and Communication Networks*, pp. 217–221, IEEE, 2011.

[17] L. Frank, D. Garcia, E. Hurley, A. Kiernan, N. Nahas, R. Walsh, and B. A. Hamilton, "Security credentials management system (SCMS) design and analysis for the connected vehicle system: Draft.," Tech. Rep. FHWA-JPO-, U.S. Department of Transportation, 2013.

[18] A. Fuchs, S. Gürgens, L. Apvrille, and G. Pedroza, "On-board architecture and protocols verification," *EVITA Project, Tech. Rep. Deliverable D3. 4.3*, 2010.

[19] A. Aijaz, B. Bochow, F. Dötzer, A. Festag, M. Gerlach, R. Kroh, and T. Leinmüller, "Attacks on inter vehicle communication systems-an analysis," *Proc. WIT*, pp. 189–194, 2006.

[20] Keen Security Lab, "Experimental Security Assessment of BMW Cars: A Summary Report," tech. rep., Keen Security Lab, 2018.

[21] Tencent Keen Security Lab, "Experimental Security Research of Tesla Autopilot," tech. rep., Keen Security Lab, Mar. 2019.

[22] M. Cesana, L. Fratta, M. Gerla, E. Giordano, and G. Pau, "C-VeT the UCLA campus vehicular testbed: Integration of VANET and Mesh networks," in *2010 European Wireless Conference (EW)*, pp. 689–695, IEEE, 2010.

[23] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security*, p. 16, 2011.

[24] C. Miller and C. Valasek, "A Survey of Remote Automotive Attack Surfaces," tech. rep., IOActive, 2014.

[25] Y. Park, J. H. Yang, and S. Lim, "Development of complexity index and predictions of accident risks for mixed autonomous driving levels," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1181–1188, IEEE, 2018.

[26] W. Zhang and W. Wang, "Learning V2V interactive driving patterns at signalized intersections," *Transportation Research Part C: Emerging Technologies*, vol. 108, pp. 151–166, 2019.

[27] F. Tanshi, K. D. Nobari, J. Wang, and D. Söffker, "Design of Conditional Driving Automation Variables to Improve Takeover Performance," *IFAC-PapersOnLine*, vol. 52, no. 8, pp. 170–175, 2019.

[28] T. Stoll, J. Imbsweiler, B. Deml, and M. Baumann, "Three Years CoInCar: What Cooperatively Interacting Cars Might Learn from Human Drivers," *IFAC-PapersOnLine*, vol. 52, no. 8, pp. 105–110, 2019.

[29] K. Gao, D. Yan, F. Yang, J. Xie, L. Liu, R. Du, and N. Xiong, "Conditional artificial potential field-based autonomous vehicle safety control with interference of lane changing in mixed traffic scenario," *Sensors*, vol. 19, no. 19, p. 4199, 2019.

[30] Z. Wang, X. Zhao, Z. Xu, X. Li, and X. Qu, "Modeling and field experiments on lane changing of an autonomous vehicle in mixed traffic," *Computer-aided Civil and Infrastructure Engineering*, 2019.

[31] S. E. Shladover, D. Su, and X.-Y. Lu, "Impacts of cooperative adaptive cruise control on freeway traffic flow," *Transportation Research Record*, vol. 2324, no. 1, pp. 63–70, 2012.

[32] F. Navas and V. Milanés, "Mixing V2V-and non-V2V-equipped vehicles in car following," *Transportation Research Part C: Emerging Technologies*, vol. 108, pp. 167–181, 2019.

[33] B. Vieira, R. Severino, E. V. Filho, A. Koubaa, and E. Tovar, "COPADRIVe - a realistic simulation framework for cooperative autonomous driving applications," in *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, pp. 1–6, Nov. 2019.

[34] The Coq Development Team, "The Coq Proof Assistant, version 8.11.0," Jan. 2020.

[35] S. Owre, N. Shankar, and J. Rushby, "Prototype Verification System (PVS)." SRI International, 1992.

[36] University of Cambridge and Technische Universität München, "Isabelle," 1986.

[37] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *CAV* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 334–342, Springer, 2014.

[38] M. Völker, M. Kloock, L. Rabanus, B. Alrifaee, and S. Kowalewski, "Verification of Cooperative Vehicle Behavior using Temporal Logic," *IFAC-PapersOnLine*, vol. 52, no. 8, pp. 99–104, 2019.

[39] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[40] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[41] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[42] R. Beers, "Pre-rtl formal verification: An intel experience," in *Proceedings of the 45th Annual Design Automation Conference*, DAC âĂŹ08, (New York, NY, USA), p. 806âĂŞ811, Association for Computing Machinery, 2008.

[43] D. Jackson, "Software abstractions-logic, language, and analysis, revised edition," *The MIT Press*, 2012.

[44] D. Jackson, "Alloy: a language and tool for exploring software designs," *Communications of the ACM*, vol. 62, no. 9, pp. 66–76, 2019.

[45] P. Zave, "Lightweight Modeling of Network Protocols in Alloy," *ACM CoNEXT*, 2010.

[46] A. Svendsen, B. Møller-Pedersen, Ø. Haugen, J. Endresen, and E. Carlson, "Formalizing train control language: Automating analysis of train stations," in *Comprail*, pp. 245–256, 2010.

[47] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed computing*, vol. 2, no. 3, pp. 117–126, 1987.

[48] D. Jackson, "Alloy: A Language and Tool for Exploring Software Designs," *Communications of the ACM*, 2019.

[49] A. Cunha, "Bounded model checking of temporal formulas with Alloy," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 303–308, Springer, 2014.