UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

Symbolic Analysis of Control Models for Autonomous Driving

Master's Thesis

Master's Degree in Software Systems Engineering and Technology

AUTHOR: Padró Ferragut, Cristina

Tutor: Alpuente Frasnedo, María

Experimental director: SAPIÑA SANCHIS, JULIA

ACADEMIC YEAR: 2021/2022

Acknowledgments

My supervisors, María Alpuente and Julia Sapiña, have been invaluable for the development of this project. Their advice and guidance have been instrumental, as well as their constant support, patience and encouragement. I am highly proud and grateful for my time working with them.

This work has been partially financed by the Generalitat Valenciana under grant *PROMETEO/2019/098 (DeepTrust)*.

Many thanks to my parents and partner for their unwavering support as well.

Resum

Es preveu que els vehicles siguen completament autònoms per a 2040, la qual cosa aportarà beneficis considerables a la societat. Els vehicles autònoms (AV) depenen en gran manera dels avanços en moltes de les tècniques i enfocaments de la intel·ligència artificial (IA), imposant nous desafiaments quant a poder garantir la seua seguretat i confiabilitat. La necessitat de protocols de control de vehicles autònoms la correcció dels quals puga verificar-se formalment és primordial per a la seguretat de tots, tant en els centres urbans com en carretera, el qual només pot abordar-se aplicant mètodes formals

Una situació representativa i de risc que es apareix amb freqüència durant la conducció ocorre quan dues o més vehicles troben una intersecció. Per a manejar aquest tipus de situacions perilloses, han d'existir uns certs protocols que puguen garantir un entorn de conducció segur, evitant bloquejos, col·lisions, inconsistències en la comunicació i altres comportaments indesitjables. En aquest escenari, la combinació de tècniques de verificació formal i raonament simbòlic pot contribuir a aportar les garanties de bon funcionament necessàries.

L'objectiu d'aquest projecte és analitzar formalment algunes propietats crítiques d'un protocol de control de vehicles autònoms recentment proposat a l'àrea. Per a aconseguir això, primer es desenvoluparà un model formal del dit protocol en el llenguatge d'especificació d'alt rendiment Maude. A continuació, es durà a terme una anàlisi simbòlic de l'alcanzabilidad basat en *narrowing* per a ajudar a descobrir qualsevol fallada eventual de seguritat en el model, de manera que es puga identificar situacions de risc i eventualment obtindre una nova versió corregida del protocol. Usant el comprobador de models de Maude, es pretén verificar les propietats generals de seguretat i vivacitat del protocol corregit, demostrant, per exemple, que no es pot aconseguir cap estat considerat insegur i que hi ha garantia de servei, és a dir, que a cap vehicle en la intersecció se li denega perpètuament progrés.

Keywords— Raonament simbòlic, Verificació formal, Narrowing en Maude, Vehicles autònoms, Intel·ligència artificial

Resumen

Se prevé que los vehículos sean completamente autónomos para 2040, lo que aportará beneficios considerables a la sociedad. Los vehículos autónomos (AV) dependen en gran medida de los avances en muchas de las técnicas y enfoques de la inteligencia artificial (IA), imponiendo nuevos desafíos en cuanto a poder garantizar su seguridad y confiabilidad. La necesidad de protocolos de control de vehículos autónomos cuya corrección pueda verificarse automáticamente es primordial para la seguridad de todos, tanto en los centros urbanos como en carretera, lo cual sólo puede abordarse aplicando métodos formales

Una situación representativa y de riesgo que se aparece con frecuencia durante la conducción ocurre cuando dos o más vehículos encuentran una intersección. Para manejar este tipo de situaciones peligrosas, deben existir ciertos protocolos que puedan garantizar un entorno de conducción seguro, evitando bloqueos, colisiones, inconsistencias en la comunicación y otros comportamientos indeseables. En este escenario, la combinación de técnicas de verificación formal y razonamiento simbólico puede contribuir a aportar las garantías de buen funcionamiento necesarias.

El objetivo de este proyecto es analizar formalmente algunas propiedades críticas de un protocolo de control de vehículos autónomos recientemente propuesto en el área. Para lograr esto, primero se desarrollará un modelo formal de dicho protocolo en el lenguaje de especificación de alto rendimiento Maude. A continuación, se llevará a cabo un análisis simbólico de la alcanzabilidad basado en *narrowing* para ayudar a descubrir cualquier fallo eventual de seguridad en el modelo, de modo que se puedan identificar situaciones de riesgo y eventualmente obtener una nueva versión corregida del protocolo. Usando el comprobador de modelos de Maude, se pretende verificar las propiedades generales de seguridad y vivacidad del protocolo corregido, demostrando, por ejemplo, que no se puede alcanzar ningún estado considerado inseguro y que hay garantía de servicio, es decir, que a ningún vehículo en la intersección se le deniega perpetuamente el progreso.

Keywords— Razonamiento simbólico, Verificación formal, Narrowing en Maude, Vehículos autónomos, Inteligencia artificial

Abstract

Vehicles are predicted to be fully autonomous by 2040, which can bring considerable benefits to society. Autonomous Vehicles (AV) heavily rely on advances in many Artificial Intelligence (AI) approaches and techniques, imposing new challenges to assure their safety and reliability, which can only be tackled by applying formal methods. The need for autonomous vehicle control protocols whose correctness can be formally verified is paramount to ensure everyone's safety, both in urban centers and on the road.

A representative, risky situation that frequently occurs while driving happens when two or more vehicles encounter an intersection. In order to handle these kinds of hazardous situations, some protocols must exist that can ensure a safe driving environment by preventing any deadlocks, collisions, communication inconsistencies, or any incorrect behaviors. In this scenario, the combination of formal verification techniques and symbolic reasoning can contribute to providing the necessary guarantees of good functioning.

The aim of this project is to formally analyze some critical properties of an autonomous vehicle control protocol that was recently proposed in this area. In order to achieve this, first a formal model of said control protocol is developed in the high-performance specification language Maude. A *narrowing*-based symbolic reachability analysis is then undertaken to help discover any eventual safety flaws in the protocol and identify risky situations so that a new, corrected version of the model can be obtained eventually. By using Maude's logical model checker (LMC), both safety and liveness properties of the corrected protocol will be eventually verified, proving that no states considered unsafe can be reached and that the system is starvation-free, i.e., that no vehicle at the intersection is perpetually denied to proceed.

Keywords— Symbolic Reasoning, Formal Verification, Narrowing in Maude, Autonomous Vehicles, Artificial Intelligence

Contents

Acknowledgments **i**

1 Introduction **1**
1.1 Autonomous Vehicles 2
1.2 Objectives 5
1.3 Related Work 5
1.4 Structure of the MSc Thesis 8

2 Preliminaries **9**
2.1 Rewriting Logic 10
2.2 The Maude Language 11
2.3 Model Checking 15
2.4 Formal reasoning in Maude 18

3 Maude Specification of an Intersection Protocol **21**
3.1 LPJL Protocol Variant 21
3.2 Model development 23
3.3 Sequences of events in the intersection 26
3.4 Intersection event characteristics 27

4 Analysis and Verification of the AV Model **35**
4.1 Classical model-checking analysis 35
4.2 Symbolic, narrowing-based analysis 41

5 Conclusions and Future Work **47**

Bibliography **49**

Appendix **53**

Chapter 1

Introduction

Many individuals are critically wounded or killed in road accidents as a result of human failures, such as driver inattention and distraction, reckless driving, or poor driving capabilities, as well as many other errors (including breaches of traffic regulations). Furthermore, traffic safety is influenced by vehicle malfunction (e.g., brake failure) and ambient factors (e.g., insufficient road information or a lack of security infrastructure).

A new form of vehicle, known as an autonomous vehicle, is being developed to increase road traffic safety by allowing a driving automation system to assist a human driver to operate the vehicle safely with improved detection and recognition, judgment, and driving abilities. Moreover, they communicate with other vehicles as well as elements of their surroundings, essentially being able to communicate with everything. As a result, once extensively deployed, automated vehicles are predicted to minimize human mistakes, improve traffic flow, and improve overall road safety and driving experience [16].

Autonomous vehicles are a burgeoning technology. Sensors are unreliable as of yet, communication between vehicles is still in its infancy and there is a too large variety of communication protocols based on IoT technologies with no clear standard with which companies should work.

Furthermore, the automotive industry advertises some vehicles as self-driving when, in fact, referring to different levels of circulation assistance. This inaccurate marketing, at its best, cultivates a lukewarm opinion of these vehicles. At its worst and when taken at face value, however, this marketing can encourage customers to trust the manufacturer's criteria when determining how the car should operate, potentially increasing the risk of an accident. Nevertheless, autonomous vehicles are on the way.

The Institute of Electrical and Electronics Engineers (IEEE) has predicted that by 2040, autonomous vehicles will account for up to 75% of all vehicles on the road. The group went even farther, predicting how infrastructure, culture, and attitudes may gradually shift by the middle of the century, when self-driving automobiles become the norm. In order to achieve this, though, a new type of *vehicle-to-vehicle* communication, as well as *vehicle-to-infrastructure* communication, must be thoroughly developed. This would allow vehicles to communicate situational data in order to prevent colliding with one another, as well as with enabling the exchange of data such as their location, destination, and intended path with a central station, which would coordinate and dispatch traffic information to route vehicles appropriately [25].

Vehicles will slowly grow their knowledge database thanks to these *vehicle-to-vehicle* and *vehicle-to-infrastructure* communications. This will allow them to know how to counter critical situations, but only when a behavior protocol has been established beforehand. Here is where control protocols come into play. Formally verifiable control protocols ensure safety, both in cities and on the road, by determining how vehicles should act. Control protocols allow us to prove correct behavior can be provided, thus making traffic safer and more efficient.

1.1 Autonomous Vehicles

Driving is one of the most performed activities of our daily lives, yet it is also one of the most hazardous. According to the World Health Organization, approximately 1.3 million people die each year worldwide as a result of road traffic accidents, with an average of 3,300 daily fatalities [26]. Many of these accidents happen due to human error:

- **Speeding**: Higher speeds are directly related to the likelihood of a crash occurring as well as the severity of the crash itself and accidents 65 km/h and above have a significantly high fatality risk.
- **Alcohol and drugs**: Driving under the influence of alcohol and psychoactive drugs drastically increases the chances of a crash occurring.
- **Poor use of safety equipment**: The lack of use of safety equipment such as helmets (for bicycles and motorcycles), seat-belts or child restraints affects dramatically the probability of fatality happening.
- **Distracted driving**: The fastest rising problem when driving is the increase of distractions while driving, especially due to using mobile phones. Drivers using their phones on the road are around 4 times more likely to be involved in a crash.
- **Unsafe vehicles and road infrastructure**: Vehicles that don't follow United Nations regulations due to poor manufacturing standards in some countries, vehicles that have not been properly upkept or don't pass safety inspections, poor road design and poor road maintenance can pose significant risks while driving and can more easily lead to traffic accidents.

Introduction

Although traffic fatalities have steadily declined over the past decade, this decrease has been due to passive crash protection optimization with systems such as airbags or seatbelts [8]. Driving assistance systems that are available on modern vehicles, which provide different levels of assistance during driving, such as speed limit detection or parking assistance, thanks to their sensors and onboard computer units, have also expedited this decline. Be that as it may, these are merely tools to account for and mitigate accidents caused by human error; but cannot truly eliminate it.

This is where autonomous vehicles (AVs) come in. AVs are not a tool to assist driving. Instead, true vehicle autonomy eliminates human error entirely since transportation is handled in full by the machine. While this technology is still in its infancy, autonomous vehicles promise to represent a good solution to make our roads safer.

Autonomous vehicles can proactively detect risks before they can occur. By using their communication protocols to obtain information about their environment, these vehicles can foresee and prevent accidents or other perilous situations by either supporting and warning a human driver or by actively taking control of the vehicle in order to avoid a crash or, if not possible, mitigate it as much as possible.

However, neither authorities nor manufacturers have established a set of protocols to follow when encountering hazardous situations, nor have they reached a consensus on what their vehicles should do when reaching critical situations such as overtaking another vehicle or arriving at an intersection. On top of that, companies can implement behaviors that imitate human behavior in such a way that tolerates traffic law violations, which is proven to be extremely dangerous.

Nowadays, vehicles are equipped with a large amount of sensors, processors and software. Some even have a new separate core platform and are equipped with a redesigned wiring distribution to have enough processing power to be “intelligent”, since traditional wiring is insufficient due to relying on mechanisms that depend on the engine, transmission, breaking and steering systems.

Nonetheless, relying purely on sensors for hazard detection is highly ineffective. Sensors may fail, a common occurrence, as a result of any number of reasons such as: normal deterioration, poor production, overuse, inadequate installation, etc. Additionally, failure can happen due to errors in the information processing unit, like not properly recognizing an obstacle if it was not in its knowledge database before, not having enough information about a situation to react or receiving erroneous information from the sensors and reacting accordingly to the situation presented by the faulty hardware. As an example, sensors may fail to detect a fence placed to block a lane, causing the vehicle to run into it or a vehicle maneuvers to overtake another vehicle fatally end up merging into a lane occupied by a larger vehicle that its protocol did not adequately consider.

All the above leads us to one of the many problems these vehicles present: the lack of protocol standardization. While more and more vehicles are becoming “intelligent”, manufacturers are still the ones responsible for designing the protocols these vehicles must follow, without requiring an established consensus with other companies. There is no regulated list of actions vehicles must follow when they encounter a critical situation.

Therefore, situations such as always stopping when a *stop* sign is found, keeping safe distances when overtaking another vehicle or knowing who has priority in an intersection are situations the manufacturer should account for. In addition, protocol design that mimics human behavior should be discouraged, as it could reinforce patterns that enable traffic law violations or otherwise allow for dangerous behavior in critical situations. This would not only put lives at risk, but could result in a vehicle recall, costing the company large amounts of money.

A relevant protocol considered in this work is the one by Lim, Jeong, Park and Lee [21], which specifies the safe behavior of autonomous vehicles in an intersection. It guarantees *mutual exclusion* while at the same time ensuring *deadlock-freedom* and *starvation-freedom*. We postpone to Section 1.3 the description of this protocol, which we use as a basis for developing a new variant of this protocol that allows us to deploy our modeling and verification techniques.

1.2 Objectives

The purpose of this MSc Thesis is to look at the key elements pertaining to a variant of the well-known autonomous vehicle control system (the *Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol* [6]) in order to formally verify the protocol and eventually propose corrections to the system. To do so, we first construct a formal model of the protocol's variant in the high-performance specification language `Maude`. Safety and liveness properties of the updated protocol model is then verified using `Maude's LTL model-checker`, confirming that no dangerous states can be reached and that the system is starvation-free, i.e. no vehicle at the crossing is refused the right to advance indefinitely. Lastly, the model is subjected to a configuration-independent, *narrowing*-based, symbolic reachability analysis to help in the detection of any potential protocol flaws so that a new, corrected version is generated, which is also capable of ensuring security properties through reachability.

1.3 Related Work

Two of the biggest technologies to achieve the effective communication from vehicles to their environment and vice-versa are the `DSRC` (based on `WLAN` infrastructure) and `V2X` (based on `5G` infrastructure) communication protocols [7]. Both protocols support communication among vehicles and allow them to communicate with their environment, but have not been fully adopted by companies and neither has been established as an industry standard.

Communication protocols such as `DSRC` and `V2X` receive data from vehicles, infrastructure or even pedestrians, informing the vehicle of everything that surrounds it and helping it map the world around it. Furthermore, the low latency of said protocols keeps the vehicle informed at all times, making it highly reliable. Thanks to these protocols, vehicles can inform each other of accidents or obstacles so as to avoid them, as well as situations such as traffic jams, redirecting incoming traffic to avoid those areas and optimize flow.

Through these communication protocols, vehicles are able to synchronize efficiently when encountering critical situations that require them to carry out certain behavior protocols. Here is where behavior protocols would come into play: stop protocols, overtake protocols, pedestrian crossing protocols, etc. With established, standardized protocols, safety would be ensured through universal verified systems. Vehicles would carry out said behavior protocols regardless of manufacturer and unwanted unsafe situations would be avoided.

The problem of coordinating multiple self-driving vehicles is a multidisciplinary endeavor that has been studied in multiple research areas (see [4] and references therein).

In the *LJPL Autonomous Vehicle Intersection Control Protocol* [21], the authors describe an intersection consisting of eight lanes of equal importance, as illustrated in Figure 1.1. Once the intersection has been entered, the vehicle or set of vehicles belonging to the same lane have exclusive access to the intersection until they have all crossed. Said protocol has been modeled several times, as seen in [18] and [6] (which was later further developed in [24]).

LJPL considers vehicles in the same lane as a unit to improve efficiency by allowing them to cross while ensuring mutual exclusion, enhancing efficiency, as well as allowing vehicles from different lanes to cross the intersection at the same time as long as they do not conflict with each other, as determined by certain parameters defined in the algorithm. It also utilizes *vehicle-to-vehicle* communication, decentralizing coordination between vehicles.

In this protocol, vehicles can take a number of actions before finishing the protocol execution:

- approach the intersection.
- stop at the intersection.
- enter the intersection (with mutual exclusivity between lanes). When the head vehicle of the lane enters the intersection, all vehicles behind it that have already stopped can enter the intersection as well.
- exit the intersection. If more than one vehicle had entered the intersection, all other vehicles from different lanes will wait until the intersection is completely free before entering. The execution ends when all vehicles have crossed the intersection.

More sophisticated, vehicle coordination with vehicular communication protocols have been devised for intelligent intersection control. In most of them, vehicles are normally assigned priorities based on their arrival times at the intersection, with vehicles reaching the intersection earlier assigned higher priorities than vehicles coming later. For instance, Azimi et al. [9] proposed several V2V-based spatio-temporal intersection protocols which are typically priority-based intersection protocols. For a detailed account of this line of work, we refer to [4].

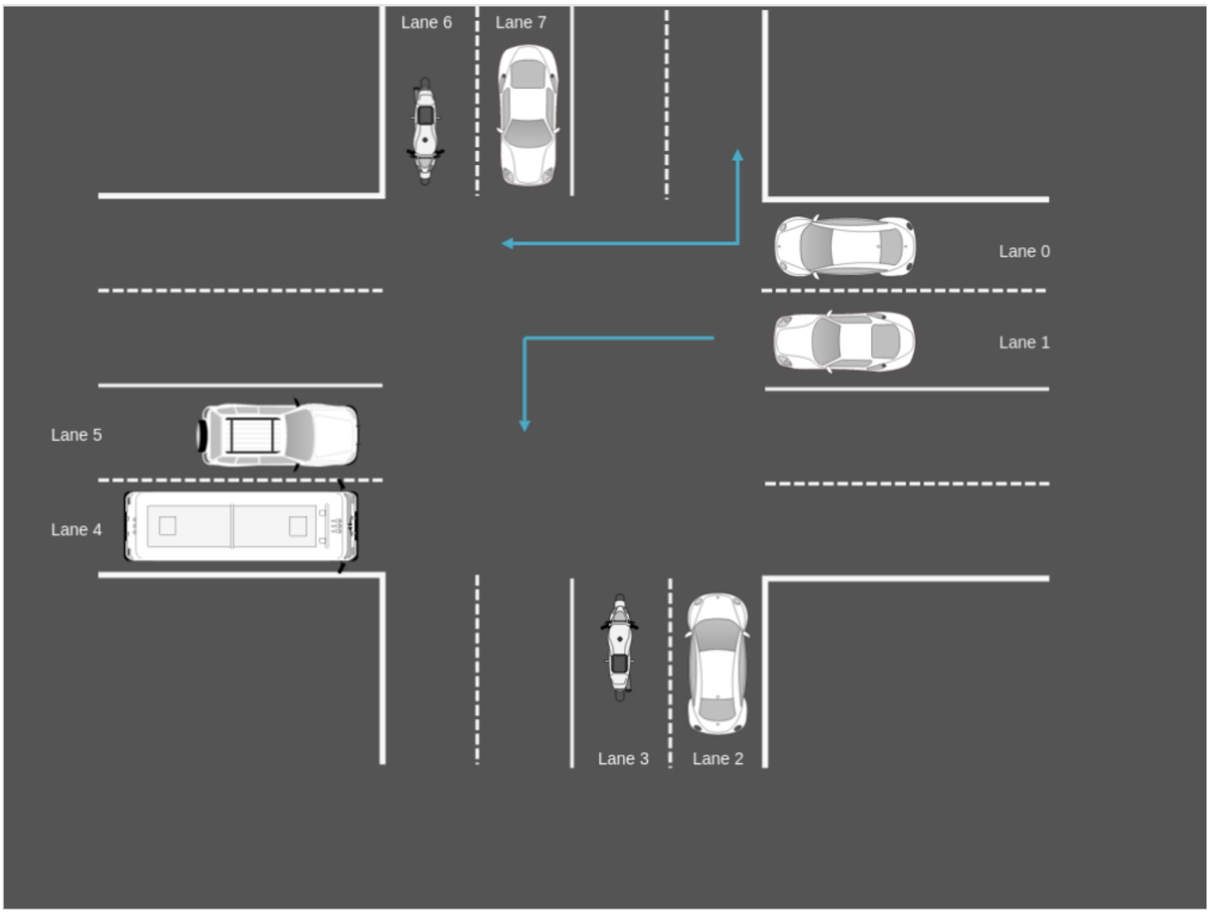


Figure 1.1: Diagram of an intersection with 8 lanes and the directions vehicles can choose to take from each lane, as described in [21]

A framework for modeling and model checking autonomous vehicles that considers the impact of communication delays by means of timed automata is proposed in [5]. More related to our work is the approach of [6], which relies on Maude's interpreter and model-checker for the analysis of the LJPL protocol.

1.4 Structure of the MSc Thesis

This thesis is organized as follows. First, we introduce the main ingredients of this work: the language and the protocols to be evaluated. The language used to represent our models, `Maude`, is briefly described in Chapter 2. Model-checking is the subject of Chapter 3. Chapter 4 gives a comprehensive description of the considered protocols, explaining the algorithms they follow. In Chapter 5, we go through the model we developed for the analysis of these protocols in great depth. The symbolic analysis and model-checking method for the created model is described in Chapter 6. Finally, in Chapter 7, we provide some conclusions and directions for future work. As an addendum, we provide an appendix which details a list of results and counterexamples given by **Maude**'s model-checker.

Chapter 2

Preliminaries

In this chapter, we start by recounting some fundamentals in order to facilitate the comprehension of the model's development and verification process: formal languages, formal methods and automated reasoning. After this, we examine all of the elements that were core to our developments. In Section 2.2 we introduce `Maude`'s core syntax and how to apply it, in Section 2.3 we briefly describe `Maude`'s model-checker and its applications and, lastly, in Section 2.4 we summarize the key ideas about term rewriting and *narrowing* in Maude.

`Maude`, being an exceedingly expressive, flexible, and high-performance language, allows for a natural representation of problems and applications. These qualities make this language invaluable as a representational device. Furthermore, both `Maude`'s interpreter and its built-in *model-checker* support for deeply thorough analysis and verification of a provided model, whether through reachability analysis or safety and liveness verification, ensuring a model fulfills all required properties.

In the most recent `Maude` versions we can efficiently use *narrowing* as well, through the interpreter itself, granting access to even more tools for an in-depth analysis of a model. The most recent `Maude` version is available at `http://maude.cs.illinois.edu`. For a more detailed description about `Maude` itself, we refer to [[13], [14],[23]].

Formal language

We can define a formal language, also simply known as language, as an infinite set of strings from a given alphabet or dictionary. Said alphabet must be a finite set of symbols. A language can also include the empty string, denoted by λ. Formal grammars and automata that can be defined as modifications of non-deterministic Turing machines serve to effectively specify a formal language [22]. Several operations can be applied on formal languages: Kleene closure, union, intersection, complement and difference [15].

A formal language, taken at some level of abstraction, can be expressed as a collection of properties a system should satisfy. This is what we can understand as a formal specification of the system behavior [20]. This specification can be verified by checking whether the design satisfies the required properties, analyzing the finite number of configurations (states) the system can reach [17].

Formal methods

Formal methods can be defined as mathematically-based rigorous techniques and tools for specifying, developing and verifying systems, both software and hardware. Formal methods can be used to assist in the elimination of errors by symbolically examining the entirety of the digital design. This allows for the establishment of correctness and safety properties that are true for all possible inputs [11].

Automated reasoning

Reasoning can be defined as the ability to make inferences [27]. Therefore, *automated reasoning* can be understood as the mechanization of reasoning through computing systems, following formal logic and, usually, deductive mathematical reasoning (although induction, abduction and non-monotonic reasoning can also be applied). By implementing an algorithmic description to a formal calculus, theorems of said calculus can be proven efficiently.

Automated reasoning can help tackle problems in mathematics and logic, solve engineering problems and is a widely used tool in computing science. When automated reasoning lays on a unification-based goal solving mechanism, automated reasoning is often referred to as symbolic reasoning.

2.1 Rewriting Logic

Rewriting logic can be defined as a *logical framework* in which we can represent many different models of concurrency, distributed algorithms, programming languages, and software and hardware as *rewrite theories* [23].

In `Maude`, a *functional module* represents a theory in membership equational logic. We can view this theory as $\mathcal{E} = (\Sigma, E \cup A)$, where Σ, the signature, specifies the type structure and theory operators, E specifies a collection of equations and A is a collection of distinguished equations that specify algebraic properties of the theory operators.

If, along with said equational theory, we consider a set R of rewrite rules that specify a number of local concurrent transitions, we can define a rewrite theory as $\mathcal{R} = (\Sigma, E \cup A, R)$.

`Maude` programs can be considered rewriting theories, each of them being a *system module*. When no rewrite rules are declared, though, the module is interpreted as a *functional module*.

2.2 The Maude Language

Maude is a multi-paradigm declarative language that efficiently implements Rewriting Logic (RWL). Rewriting logic allows for concurrent systems to be expressed in a simple and effective way by defining them as a set of terms that define states, equations that simplify said states and rules that define how those states can be transformed independently and non-deterministically.

Additionally, `Maude` is among the fastest equational rewriting systems in its class. Applications for this programming language include `Maude`'s metalanguage, theorem provers, languages, and models of computation [19].

`Maude` supports both equational and rewriting logic specifications. `Maude`'s basic programming statements are *equations* and *rules*, both having simple *rewriting semantics* that allow for replacements from any instance of a left-hand side pattern to a corresponding instance of the right-hand side pattern of any rule or equation.

Core Maude, Syntax and Basic Parsing

Modules are the main elements that comprise a specification. `Signatures` and `statements` make up said modules. In `Core Maude` we find two different types of modules, the distinction between these two being the types of statements they can host: *functional modules*, admit *equations* while *system modules* also admit *rules*. `Signatures` are the basic syntax declaration part of a module, whereas `statements` are made up of `equations` and `rules`. Both of these allow the programmer to assert the behavior of a given system. The syntax declaration includes:

- *sorts and subsorts*: allow us to name types of data and organize them hierarchically.
- *kinds*: a more general declaration of a *type*. It is typically used for error handling, as it can handle partially evaluated expressions (we will not go over `kinds` in this chapter, as they were not used in our model).
- *operators*: are used to create operations that can be applied to the data to build expressions and functional applications.

Identifiers are the basic syntactic elements that constitute a `Maude` program and are used to name *modules*, *sorts* or form *operator* names and such. *Identifiers* are any finite sequence of ASCII characters that (1) do not contain white space, (2) are not considered `special` characters (such as ‘`{`’, ‘`}`’, ‘`[`’, ‘`]`’, ‘`(`’, ‘`)`’ and ‘`,`’) and are not the *backquote* (`` ` ``) character which is used as an *escape character* to express that a blank space or special characters do not break the sequence. Additionally, *backquotes* can *only* appear immediately *before* any of the special characters or *between* nonempty strings of characters. As an example, let us consider the *identifiers* `Vehicle`, `Bus`, `Motorcycle` and `Car`.

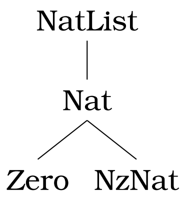
Sorts allow users to declare the types of data they require by using the keyword `sort`, followed by an identifier. They are the first element of a *module* to be declared. In order to partially order *sorts*, we can use the keyword `subsort`, expressing what data types are a *subtype* of another, similar to type hierarchies or a parent-child relation.

The `sort` declaration is structured as follows. Here we can see how a single or multiple *sorts* are declared, as well as how a `subsort` hierarchy can be defined:

```
sort <Sort> .
sort <Sort-1> ... <Sort-k> .
subsort <Sort-1> < <Sort> .
subsorts <Sort-2> ... <Sort-k> < <Sort> .
```

Following this structure, we can continue with our vehicle example by declaring the relevant problem *sorts*, which is graphically represented in Figure 2.2 as an acyclic graph:

```
sort Zero .
sorts NzNat Nat NatList .
subsorts Zero NzNat < Nat .
subsort Nat < NatList .
```



An *operator* is a user-defined structure that is comprised of a list of *sorts* and produces a new term of a *sort* as a result, that can be different from the *sorts* in the list. *Operators* may have arity zero or higher. When arity is zero, following equational theories, we can consider the *operator* a constant. Likewise, *operators* can be parameterized by using the *underscore* (_) character, indicating each parameter's location within the notation. This parameterization allows for prefix, suffix or infix notation.

If an *operator*'s purpose does not entail transformation by equations (except for axioms between *constructors*) or rewrite rules, they are considered `constructors`, meaning they are used to organize information and compose simpler terms into more complex ones. *Constructors* are indicated by means of the attribute `ctor`.

The `operator` declaration is structured as follows:

```
op <OpName> : <Sort-1> ... <Sort-k> -> <Sort> [<OperatorAttributes>] .
ops <OpName-1> ... <OpName-n> : <Sort-1> ... <Sort-k> -> <Sort> [<OperatorAttributes>] .
```

Where `<Sort-1> ... <Sort-k>` are the sorts provided as arguments for the operator, `<Sort>` is the type of result and `[<OperatorAttributes>]` are the possible attributes the operator might have and express algebraic properties such as the *associativity* or *commutativity* properties. We can also declare various operators at the same time that share the same structure using the keyword `ops`. Operators in Maude can be overloaded.

Following this structure, we can continue with our vehicle example by declaring our *operators*:

```
op zero : -> Zero .
op s_ Nat : -> NzNat .
op __ : Nat Nat -> NatList .
```

Variables are usually declared using the keyword `var`, and are constrained to being a particular *sort* or *kind*. When declared this way, their scope is the entire module. That being said, they can also be declared on-the-fly, which is used as `<varName>:<Sort>` due to its scope being the declaration's occurrence. Variable's names can be both in upper and lower case, but it is more customary in Maude to declare them in uppercase.

We can observe all the different ways variables can be declared:

```
var VarName : Sort .
vars VarName1 VarName2 : Sort .
var VarName : [Kind] .
vars VarName1 VarName2 : [Kind] .
rl [RuleName] OpName(VarName:Sort) .
rl [RuleName] OpName(VarName:[Kind]) .
```

Therefore, if we wanted to create variables for our example, they would look something like this:

```
var N : Nat .
vars N1 N2 : Nat .
var NList : NatList .
rl [RuleName] OpName(N:Nat) .
```

Functional modules and equational theories

Functional modules can be understood as aggregations of *operators* and *equations* that describe the program. When a Maude program is only written as a group of *functional modules*, it can be considered equivalent to an *equational theory*. In Maude, functional modules are assumed to have the capacity to apply its equations repeatedly until reaching the *canonical* (irreducible) form.

Functional module's key elements are *equations*, which can be declared with either the keyword eq or the keyword ceq, depending on whether they are used to declare unconditional or conditional *equations* respectively. A condition in an *equation* can be either a single equation or a conjunction of equations using the binary conjunction connective /\ which is assumed to be associative [13].

The keyword eq serves to declare unconditional equations. Both terms from the equation must be of the same kind and any variable in RTerm must also appear in LTerm. Equations are written as:

```
eq (LTerm) = (RTerm) [(Attributes)] .
ceq (LTerm) = (RTerm)
    if (Condition-1) /\ ... /\ (Condition-n)
    [(Attributes)] .
```

Axioms are properties about a given set, determined by an *operator* when defining a said set. They allow Maude to efficiently use equations. Maude currently supports associativity, commutativity, idempotency, identity element and left or right identity elements. Declaring these attributes for an *operator* is equivalent to declaring the corresponding equations for the *operator*. We can observe an example of these axioms being declared when we describe *operators* in Section 2.2.

System modules and rewrite theories

Rewrite theories with sorts, kinds, and operators are specified in system modules, as are three types of statements: equations, memberships (i.e., a distinguished kind of equations that are used to introduce type constraints), and rules, all of which can be conditional.

System modules contain equations *and* rules that specify state transitions and change a state when it matches the left side of a rule, transforming the state according to the right-hand side of the rule.

The keyword `rl` serves to declare unconditional rules. Both terms from the rule belong to the same kind. As already stated, rules are used to describe *local concurrent transitions* in a system. Rules are written as follows:

```
rl [(Label)] : (LTerm) => (RTerm)
[(Attributes)] .
```

The keyword `crl` serves to declare conditional rules. A condition can be either a single equation, membership, rewrite or rewrite expression or a conjunction of them using the binary conjunction connective `/\` which is assumed to be associative [13]. The general structure of conditional rules has the following syntax:

```
crl [(Label)] : (LTerm) => (RTerm)
    if (Condition-1) /\ ... /\
    (Condition-n)
    [(Attributes)] .
```

2.3 Model Checking

Formal methods can be understood as "applied mathematics for modeling and analyzing systems". Using mathematical logic, we can demonstrate that a program's behavior fulfills the required specification, thus proving correct behavior in an accurate manner. Formal methods, which use mathematical rigor to establish system correctness, are widely used for software verification. Verification techniques that utilize models are based on mathematically precise and unambiguous models that describe the intended system behavior.

Traditional model-checking is a flexible technique that allows for the verification of concurrent, distributed and reactive models, thus enabling an automatic and exhaustive exploration all of a system's states by brute-forcing all possible system states in a systematic, possibly implicit manner [10]. It allows the user to verify both software and hardware requirements by obtaining a set of system requirements or design (the models) and the verifiable properties that the system must sate. This verification technology, through the use of Temporal Logic (TL) formulas, can determine if a model satisfies certain required properties [12]. Furthermore, model-checking is currently the lightest and fastest verification technique available, as well as being fully automated.

Model-checking has an edge over other types of verification. Experimentation methods such as simulation and testing cannot be fully comprehensive and exhaustive, allowing for vulnerabilities in the program to go undetected that could be found and corrected through model-checking. However, model-checking's completeness can only be guaranteed in a system with a huge, yet finite number of states (> 10^{120}).

When a model does not satisfy its required properties, model-checking tools can generate counterexamples to help pinpoint why the model does not satisfy the specification [29]. To deal with systems with infinite states, the use of abstraction techniques is required.

Expressiveness and *efficiency* are important criteria for logic. *Safety* properties, *liveness* properties, *fairness* properties and *security* properties, for instance, are examples of characteristics that can and cannot be captured by logic [1]. It is critical for all required properties to have the capacity to be expressed, else this verification method would prove futile. Efficiency, on the other hand, refers to the difficulty of a logic's model-checking problem as well as the performance of the logic's model-checking algorithms.

Verifiable model-checking properties can be classified into different types, in accordance with the aspects of a system they correspond with:

- **Reachability properties**: These are the properties that guarantee the occurrence of a desired state.
- **Safety properties**: are those that ensure that a potentially harmful situation will not occur, which is the polar opposite of reachability.
- **Liveness properties**: They ensure system progress by ensuring that a process does not stop for no reason, that no process dies of starvation, and that no deadlocks occur.
- **Fairness properties**: Ensure that a property receives attention an infinite number of times, either periodically or from a specific state. This includes *recurrence* (a property occurs every so often); and also *persistence* (a property is infinitely given and maintained).

[1]Actually, security is considered to be a *hyperproperty*, as it cannot be expressed in trace-based specifications languages [1]

Temporal Logic

The ability for a logic to specify properties on infinite execution paths is a must-have in order for a model-checker to verify formulas. This is where temporal logic enters the picture. A finite state system can be realized if a program can be specified in TL. This sparked the concept of model-checking, which involves determining whether a finite state graph is a model of a TL specification. In order to avoid conforming to the Hoare-style paradigm, such systems should ideally exhibit nonterminating behavior. They are also usually interactive, distributed, and nondeterministic, i.e., a reactive system [12].

Temporal logic is a formalism for describing change over time that is used to manage the system's executions and fairness issues in order to ensure that it is correct. It adds modalities to propositional or predicate logic that allow for referral to a reactive system's infinite behavior, allowing for the specification of the relative order of events [10]. The Linear Temporal Logic (or LTL), introduced by Pnueli [28], is the temporal logic we use in this project.

Temporal logic is useful to express certain properties we are interested in having in our model. We can control different states of execution, ensuring no undesired states are found. While some of these properties could be verified through searches, the rigorous nature of model-checking through the formulation of LTL properties is a thorough approach that ensures correct behavior of the model and can assist in the discovery of problems on how the model was constructed.

LTL is not advisable for asynchronous systems, as it represents a singular timeline in which several different futures are possible depending on which component is chosen to evolve. LTL uses the logical operators: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, `true` and `false`. In addition, LTL uses temporal operators such as:

- `G`: is used for *always*, indicating that `p` is globally true.
- `F`: is used for *finally*, indicating that `p` is going to be fulfilled at some future time.
- `X`: is used for *next*, indicating that `p` will be fulfilled in the next state.
- `U`: is used for *until*, indicating that `p` must be fulfilled until `q` is fulfilled, which must be done at some point.
- `R`: is used for *release*, that `q` must be true until and also in the state in which `p` is fulfilled; if `p` is never satisfied, `q` will remain infinitely true.
- `W`: is used for *weak until*, indicating that `p` must be fulfilled at least until `q` is fulfilled; if that never happens, `p` will remain infinitely.

Model-checking in Maude

Maude's LTL model checker allows algorithmic verification of competing models that are expressed as rewrite theories. In any Maude system module, we can differentiate two levels of specification:

- **System specification level**: the rewriting theory that defines the behavior of the system.
- **Property specification level**: one or more properties of interest to be verified.

To investigate the system specification, it must be run in the Maude environment to see if it behaves as expected and intended. Property verification, on the other hand, necessitates both a logic specification and a procedure that allows them to be verified in a finite range of states.

The LTL syntax is specified in the `model-checker.maude` file. To carry out property verification, Maude uses the modules defined in this file.

Maude's LTL model checker allows you to verify:

- **Reachability properties**: They ensure that a certain state will be reached.
- **Security properties**: They guarantee that a certain configuration will not be given.
- **Liveness properties**: They ensure that an action will have its reaction.

Despite the fact that fairness can be expressed in LTL, this model checker is unable to verify fairness properties, which guarantee that a situation will repeat indefinitely. This is due to the fact that it can only work with a finite number of specific states and thus is unable to make an equational abstraction to determine whether these properties will be fulfilled in the future.

2.4 Formal reasoning in Maude

Rewriting is used in the fields of mathematics, computer science and logic to describe a variety of formal reasoning approaches that heavily rely on replacing subterms in a formula with other terms. It is possible for rewriting to be non-deterministic. One rewriting rule could be applied to a term in a variety of ways, including different subterms and the fact that more than one rule can be relevant to rewriting a specific position or more than one rule could be relevant. Rewriting systems then provide a set of possible rule applications rather than an algorithm for transforming one term to another. Rewrite systems, on the other hand, can be considered as computer programs when combined with the right algorithm, and term rewriting is used in numerous theorem provers and declarative programming languages.

Term rewriting systems are reduction systems whose objects are terms, which are rewritten according to a set of rules. Term rewriting system rules are usually written as $l \to r$ (indicating that in every possible context, any term matching the left-hand side can be replaced by the corresponding instance of the right-hand side), where both l and r are terms, l is not a variable, and every variable from r also exists in l.

A $l \to r$ rule can be employed for a term t, simplifying the term to $r\sigma$ if σ is a matcher for t and l.

Unification is the algorithmic process of solving equations between symbolic expressions. Depending on which terms appear in a set of equations and which expressions are considered equal, many frameworks of unification are determined. If higher-order variables are permitted in an expression, the process is referred to as higher-order unification. If a solution is required to make both sides of each equation modulo a set of equations, it is called equational unification. If not, it is called equational unification. A solution of a unification problem is denoted as a substitution. A unification algorithm should compute a complete and minimal set of solutions for a given problem.

Narrowing allows the use of logical variables in terms and extends term rewriting by replacing pattern matching with unification. Therefore, when given a term with variables, it ascertains the most general instances of that term that can be rewritten with existing rules.

Multi-paradigm languages such as `Maude` are capable of *narrowing*. *Narrowing* is the operational principle that integrates functional and logic programming features in a single computational paradigm by enabling the logic-like evaluation of expressions containing uninstantiated logic variables. Its original purpose was theorem proving, as it serves as a mechanism for solving equational unification problems but has found an important number of applications in many areas such as program analysis, verification, synthesis, and transformation.

Similarly to rewriting steps, with each *narrowing* step we must choose a subterm of the term t and an *unconditional* rule [2] with an instantiation of the variables of t and the rule's left-hand side. This is because, unlike rewriting steps, *narrowing unifies* the left-hand side l of the chosen subterm of t before using the selected rewrite rule in *R*.

The *narrowing* infrastructure in Maude is a quite sophisticated, three-layer machinery: 1) *narrowing* with the rules of R modulo equations E and axioms A; 2) equational *narrowing* with the equations of E (oriented from left to right as rewrite rules) modulo the axioms A; unification modulo A. For a gentle description, we refer to [2].

[2]Conditional *narrowing* is not natively supported in `Maude` yet.

Chapter 3

Maude Specification of an Intersection Protocol

In this chapter we focus on describing the formal model of an intersection protocol that has been specified in the Maude language. This model follows the protocol described in the following Section 3.1, which is based on the protocol described in Section 1.3, having an initial state represented by five different parts: a finality state, an intersection state, a clock, a set of lanes and a set of vehicles. The number of lanes in the set is six, and the number of vehicles in the set can vary, depending on the situation we desire to execute.

The first part of this chapter will break down the protocol, while the following sections describe the different parts that constitute the model, explaining what each element represents within the protocol and how they are used.

3.1 LPJL Protocol Variant

In this work, we propose a variant of the LPJL protocol for the case when a vehicle encounters an intersection that has six lanes: four lanes that comprise the main road and two secondary lanes that merge into the four main ones. This variation, while having less lanes, adds the notion of priority.

In the original protocol, all lanes are considered of equal importance. While that is a useful scenario, the concept of a lane having priority over another cannot be determined and is instead dependent on order of arrival, in which the first vehicle that arrives at the intersection is the one that determines who should have priority, in accordance with to traffic laws. In this protocol, however, this only happens with lanes that are on the same priority, but secondary lanes have modified behavior that prevents vehicles from entering the intersection if there are priority vehicles nearby. As we can see in Fig 3.1, these six lanes allow vehicles to enter the intersection either sequentially or simultaneously (as long as they do not conflict with each other).

Also, depending on the lane, each vehicle has several paths they can take, which determines who they conflict with. Secondary lanes always conflict with primary ones and vehicles that circulate through them must always wait before entering the intersection, allowing priority vehicles to circulate before them. Meanwhile, this protocol still allows simultaneous entry when lanes do not conflict with each other as well as keeping maintaining mutual exclusion for when they do conflict.

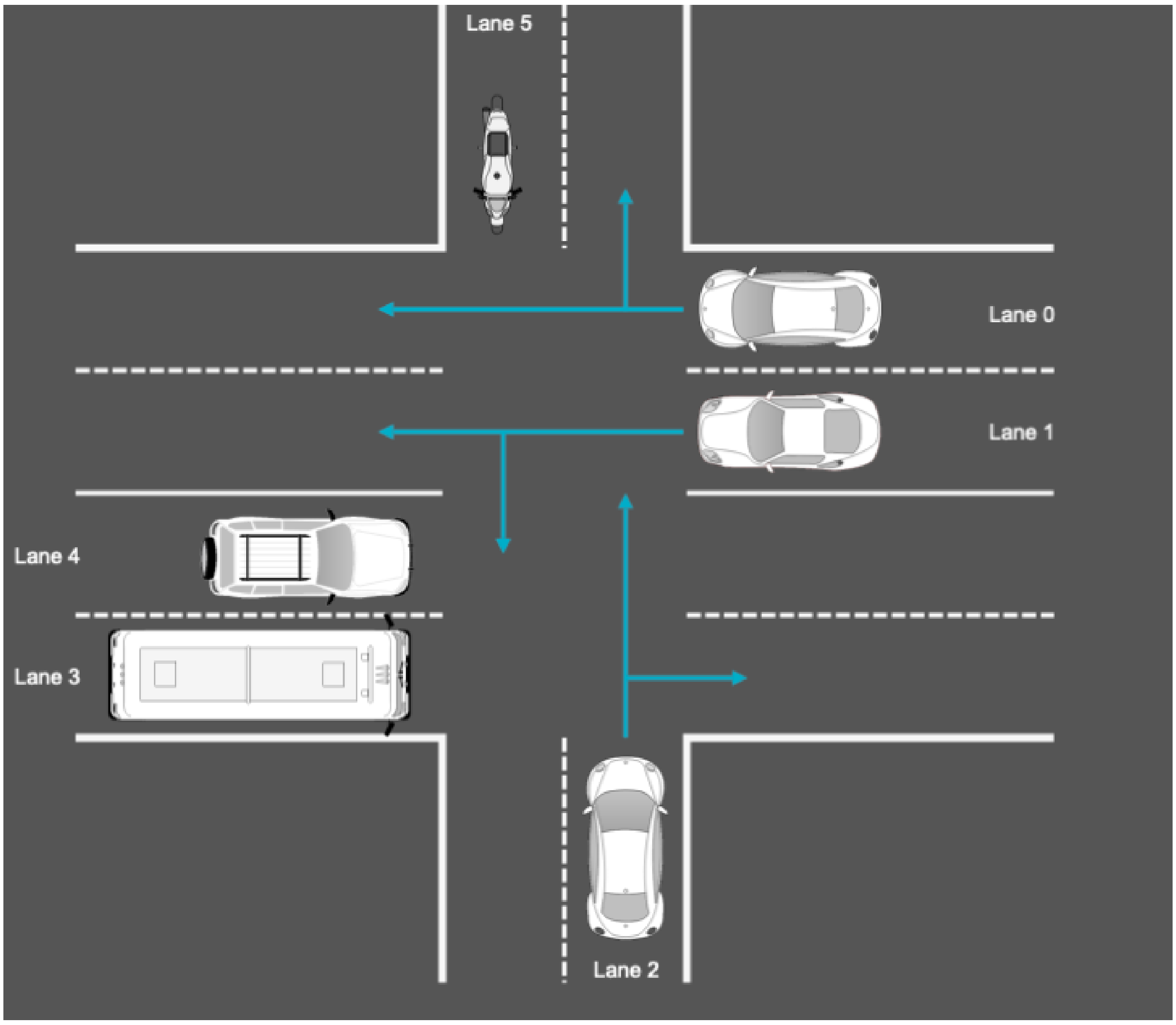


Figure 3.1: Diagram of an intersection with 6 lanes and the directions vehicles can choose to take from each lane.

The conflicting lanes that we choose for this type of intersection are:

- **Lane 0**: Conflict with Lane 2 and Lane 5.
- **Lane 1**: Conflict with Lane 2, Lane 3, Lane 4 and Lane 5.
- **Lane 2**: Secondary lane that conflicts with all lanes except Lane 5.
- **Lane 3**: Conflict with Lane 2 and Lane 5.
- **Lane 4**: Conflict with Lane 0, Lane 1, Lane 2 and Lane 5.
- **Lane 5**: Secondary lane that conflicts with all lanes except Lane 2.

Although this proposal follows the same basic actions as the *LPJL* protocol, the distinction between primary and secondary lanes allows for a representation of priority that the original protocol lacks, as all lanes are equal priority. This gives us new scenarios such as a vehicle having already arrived at the intersection but, since it belongs to a secondary lane, it must now not only check whether the intersection is empty, but also check if there are vehicles with higher priority since they must enter the intersection first.

This variation of the *LPJL* protocol allows for vehicles with the same priority to enter the intersection simultaneously as well, as long as they are not considered to be in conflict. In Figure 3.1 we can see the paths vehicles can intend to have, thus allowing us to visualize when there is or isn't a conflict between lanes.

Seeing that lanes 0, 1, 3 and 4 are on the same priority level and given the above explained list of conflicts between lanes, we have three possible combinations of lanes that allow for simultaneous access to the intersection without risk of collision: *L0-L1*, *L0-L3* and *L3-L4*. On the other hand, if there does exist a conflict between lanes, they will enter the intersection in sequential order. Lastly, vehicles from secondary lanes (Lanes 2 and 5) will wait until all other vehicles in the high priority lanes have crossed before entering the intersection.

3.2 Model development

We have specified in Maude a formal model of the considered protocol that is structured as follows. For more details, the code to this model can be found at `https://github.com/SilverArrow23/TFM`. Firstly, this model is comprised of several modules involved in vehicle control.

- `NATURAL-LIST` contains the definition of `NatList` and `NeNatList`. This is an auxiliary module, as using the normal `NAT-LIST` module generated some problems at the meta-level.
- `FSTATE` contains the definition of `State`.
- `VSTATUS` contains the definition of `VStat`.

- INTERSECTION contains the definition of all the main elements that belong to an intersection.
- INTERSECTION-EQ contains the equations with which states can be simplified.
- INTERSECTION-RL contains the rules with which the execution states will progress.
- CONFIG contains the definition the initial configuration for the execution.

Within these modules, we can find all the sorts that define the elements of the soup that represents the intersection.

- FStatus is used to indicate the status of the execution: ended or not ended.
- FStat allows us to represent a global status of the execution, indicating whether all vehicles have crossed or not.
- VStat is used do indicate the status of a vehicle.
- IStat allows us to represent a global status for the intersection, indicating whether it is currently occupied or not.
- Clock equates to a global clock for all vehicles to permit synchronization.
- Id serves as a way of identifying vehicles and lanes.
- Time allows us to represent the time at which a vehicle arrives at an intersection.
- Lane represents the structure of a lane.
- LaneSet represents the group of lanes that belong to the intersection.
- Vehicle represents the structure of a Vehicle.
- VehicleSet represents the group of Vehicles involved in the intersection during the execution.
- Config represents the configuration of the model.

Along with these sorts, we have also defined their constructors as well as their different algebraic properties:

```
*** Constructors ***
ops nEnd end : -> State [ctor] .
ops circulating stopping stopped waiting crossing crossed : -> VStat [ctor] .
op (fstat:_) : FStatus -> FStat [ctor] .
op (istat:_) : Bool -> IStat [ctor] .
op (clock:_,_,_) : Nat Bool Bool -> Clock [ctor] .
op none : -> LaneSet [ctor] .
op (l[_]:_) : Id NatList -> Lane [ctor] .
op _;;_ : LaneSet LaneSet -> LaneSet [assoc comm id: none] .
op none : -> VehicleSet [ctor] .
op (v[_]:_,_,_,_) : Id Nat VStat Time Time -> Vehicle [ctor] .
op _;;_ : VehicleSet VehicleSet -> VehicleSet [assoc comm id: none] .
```

Finality Status

The indicator of the finality status (to check whether all vehicles have crossed the intersection or not) has the following structure:

```
op (fstat:_) : FStatus -> FStat [ctor] .
```

- `FStatus`: Status of sort `FStatus`, which indicates if execution has reached its end.

Intersection Status

The indicator of the intersection's status (if it is currently occupied or not by a vehicle) has the following structure:

```
op (istat:_) : Bool -> IStat [ctor] .
```

- `Bool`: Boolean flag to indicate if its occuped (`true`) or not (`false`).

Clock

Represents ticks of time in a clock that is used to synchronize all elements of the intersection.

```
op (clock:_,_,_) : Nat Bool Bool -> Clock [ctor] .
```

- `Nat`: Natural number that represents the tick of the clock.
- `Bool`: Boolean to indicate whether the clock can be ticked or not.
- `Bool`: Boolean to indicate whether the execution can end or not.

Lane

Represents the lanes that form the intersection.

```
op (l[_]:_) : Id NatList -> Lane [ctor] .
```

- `Id`: Natural number that represents the Id of the lane. Thanks to Ids, we can know if a lane is prioritary or not in the representation of the intersection.
- `NatList`: List of Ids of the vehicles that are currently in the lane.

Vehicle

Represents the vehicles that will enter and exit the intersection.

```
op (v[_]:_,_,_,_) : Id Nat VStat Time Time -> Vehicle [ctor] .
```

- `Id`: Natural number that represents the Id of the vehicle.
- `Nat`: Natural number that represents the lane the vehicle is circulating in.
- `VStat`: Vehicle status (`circulating`, `waiting`, `crossing`, etc).
- `Time`: Time at which the head vehicle of the lane has arrived.
- `Time`: Time at which the vehicle has arrived at the lane.

Config

Once we have seen what all the different parts of the model are for, we can proceed to describe the structure of an initial state. The initial state is defined as follows:

```
sort Config .
op {_|_|_|_|_} : FStat IStat Clock LaneSet VehicleSet -> Config [ctor] .
```

An example of an initial state is as follows:

```
op init-inter : -> Config .
eq init-inter = {
    (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
    (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
    (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
    (v[0]: 0, circulating, 100, 100) ;;
    (v[1]: 0, circulating, 100, 100) ;;
    (v[2]: 0, circulating, 100, 100) ;;
    (v[3]: 0, circulating, 100, 100)
} .
```

In this state we initialize the Finality Status as `nEnd` (meaning the execution has not ended), the Intersection Status as `false` (no vehicles are occupying the intersection), Clock is at 0 and time cannot advance and the execution cannot end, all lanes are empty and five cars are circulating towards the intersection.

3.3 Sequences of events in the intersection

Before diving into the different rules and equations that comprise this program, let us first define all the actions that can occur in the intersection.

1. **Arrival**: When a vehicle arrives at the intersection, a tick of the clock is triggered to assign the vehicle a timestamp, and its state is set to `stopping`, which is intended to slow the car down before entering the intersection.

2. **Stop**: Once a car has entered a lane that belongs to the intersection (in Arrival) and has slowed down, it must stop at the intersection. There are two positions when stopping at the intersection, which are either at the entrance to the intersection (the vehicle is the head of the list of vehicles) or behind a vehicle that arrived before it.
3. **Enter**: When the vehicle has stopped, several checks are made to determine whether it can enter the intersection or not.
   - The vehicle is in a priority lane and no other vehicles are in the intersection: it can enter.
   - The vehicle is in a priority lane and another vehicle is in the intersection, but there is no conflict between these lanes: it can enter.
   - The vehicle is in a priority lane and another vehicle is in the intersection, and there is conflict between these lanes: it must wait to enter.
   - The vehicle is in a secondary lane: it must wait before entering.
   - The vehicle is in a priority lane and has waited until the intersection is unoccupied: it can enter.
   - The vehicle is in a secondary lane and has waited until all nearby priority vehicles have crossed before: it can enter.
4. **Exit**: Once the vehicle has crossed the intersection, the action to exit will take place. This "eliminates" the vehicle from the intersection, since it has completed its operation.

3.4 Intersection event characteristics

Now that we have presented the initial configuration and the different actions that can take place in the considered circulation scenario, we can proceed to explain some of the rules and equations used to transition correctly to the desired states.

Knowing that rules enable state transitions, all rules echo the formerly explained events. Furthermore, noticing that since the left-hand side of the rule corresponds to the current state and the right-hand side is the resulting state, we can tightly control when vehicles trigger certain events. This helps us avoid unwanted or unsafe states, which is further explored in Chapter 4.2.

Arrival

This is the first set of rules that can be applied. Depending on whether the vehicle arriving is the first vehicle in the lane or not, we have two different rules. This rule serves to transition the state of the vehicle `circulating -> stopping`, in which the vehicle will start to slow down before arriving at the intersection. As an example, we present the first rule, where the lane is empty upon arrival:

```
--- Arrive: they make the vehicle go from CIRCULATING to STOPPING. Arrive at intersection
--- Two transitions:
--- Head vehicle in the lane (lane is empty upon arrival)
```

```
rl [arrive1] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, B) | LSet ;; (l[LId]: nil) | VSet ;; (v[
    VId]: LId, circulating, 100, 100) }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, true, B) | LSet ;; (l[LId]: VId) | VSet ;; (v[
    VId]: LId, stopping, T, T) } [narrowing] .
```

Stop

For this transition between *stopping* and *stopped*, we make use of a rule which stops the first vehicle of the set that can stop. This ensures all vehicles stop in order of arrival relative to their clocks, due to it being nonsensical for a vehicle to stop at the intersection before the vehicle that is directly in front of it:

```
--- Stop: it makes the vehicle go from STOPPING to STOPPED. Stop at intersection
--- One transition:
rl [stop1] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, B) | LSet | VSet ;; (v[VId]: LId,
    stopping, VT, VT') }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, B) | LSet | stopVehicle((v[VId]: LId,
    stopping, VT, VT') ;; VSet) } [narrowing] .
```

This rule uses the equations `stopVehicle` and `isMinStop` to determine which is the first vehicle that should stop and returns the modified set:

```
op stopVehicle : VehicleSet -> VehicleSet .
eq stopVehicle(none) = none [variant] .
eq stopVehicle((v[VId]: LId, stopping, VT, VT') ;; VSet) =
    if isMinStop((v[VId]: LId, stopping, VT, VT'), VSet)
    then (v[VId]: LId, stopped, VT, VT') ;; VSet
    else (v[VId]: LId, stopping, VT, VT') ;; stopVehicle(VSet)
    fi [variant] .


op isMinStop : Vehicle VehicleSet -> Bool .
eq isMinStop((v[VId]: LId, stopping, VT, VT'), none) = true [variant] .
eq isMinStop((v[VId]: LId, stopping, VT, VT'), (v[VId']: LId', VSt, VT2, VT2') ;; VSet) =
    if VSt == stopping
    then
        if VT' < VT2'
        then true and-then isMinStop((v[VId]: LId, stopping, VT, VT'), VSet)
        else false
        fi
    else isMinStop((v[VId]: LId, stopping, VT, VT'), VSet)
    fi [variant] .
```

Enter

In order to enter the intersection, the model checks two important properties: if the vehicle is first in line and if there is already a vehicle in the intersection.

- If there are no cars in the intersection and it is a vehicle in a priority lane, it can enter directly. If it is not a priority vehicle, it must wait until all present priority vehicles have already crossed.
- If there is already a vehicle in the intersection but there is no conflict between lanes, it can enter the intersection as well.

- If no other criteria has been met, the vehicle will wait. If it is a priority vehicle it will enter once the intersection is empty. Otherwise, it will make sure all priority vehicles have already crossed.

In this rule we have the first state it can match with, which is when a vehicle that has stopped encounters an empty intersection:

```
rl [enter1] :
{ (fstat: nEnd) | (istat: false) | (clock: T, B, B’) | LSet ;; (l[LId]: VList) | VSet ;; (v[
     VId]: LId, stopped, VT, VT) }
=>
if isPriorityLane(LId)
then
    { (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId]: VList) | (v[VId]:
         LId, crossing, VT, VT) ;; letCross(VList, VSet) }
else
    if canSecondaryEnter(LSet)
    then
        { (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId]: VList) | (v[VId
            ]: LId, crossing, VT, VT) ;; letCross(VList, VSet) }
    else
        { (fstat: nEnd) | (istat: false) | (clock: T, B, B’) | LSet ;; (l[LId]: VList) | VSet
            ;; (v[VId]: LId, waiting, VT, VT) }
    fi
fi [narrowing] .
```

To make sure a vehicle is in a priority lane, we use the isPriorityLane equation, which checks the ID of the lane to see if it matches any of the priority lanes.

Once a lead car is crossing the intersection, all vehicles that are behind and have already stopped are also allowed to cross. This is done with the equation letCross which checks if any vehicles match with the current stopped vehicles in that lane and flags all of them to cross. The specification of the letCross operator looks like this:

```
op letCross : NatList VehicleSet -> VehicleSet .
eq letCross(VList, none) = none .
eq letCross(VList, ((v[VId]: LId, VSt, VT, VT’) ;; VSet)) =
    if VId in VList and VSt == stopped then (v[VId]: LId, crossing, VT, VT’) ;; letCross(VList
        , VSet)
    else (v[VId]: LId, VSt, VT, VT’) ;; letCross(VList, VSet)
    fi [variant] .


op _in_ : Nat NatList -> Bool .
eq VId in nil = false .
eq VId in (H:Nat T:NatList) = VId == H:Nat or-else
    VId in T:NatList [variant] .
```

where _in_ is an auxiliary equation that checks if a number is in a list of numbers.

If it is not a priority lane, canSecondaryEnter instead checks if all other nearby primary vehicles have already crossed:

```
op canSecondaryEnter : LaneSet -> Bool .
eq canSecondaryEnter(none) = false .
eq canSecondaryEnter((l[0]: VList) ;; (l[1]: VList1) ;; (l[3]: VList2) ;; (l[4]: VList3) ;;
     LSet) =
    if VList == nil and VList1 == nil and VList2 == nil and VList3 == nil
    then true
    else false
    fi [variant] .
```

On the other hand, if the intersection is not empty, it will instead check if the vehicle that is requesting entry is not in conflict with the vehicle within the intersection. If it is not and it is a priority vehicle, it will be able to enter the intersection as well. Furthermore, being a priority vehicle, it must check if the lane it is in is the *edge* type or the *center* type, as their behavior is different (as illustrated in Figure 3.1). If, on the other hand, there is a conflict or it is not a priority vehicle, it will be set to the *waiting* state instead:

```
--- Check if a number is even or odd
op isEdge : Nat -> Bool .
eq isEdge(LId) =
    LId == 0 or
    LId == 3 [variant] .


--- Intersection has car. Check to see if lanes are prioritary and in conflict
---- If prioritary and no conflict -> enter the intersection immediately
---- If not prioritary or has conflict -> wait until all other prioritary vehicles have
     crossed
rl [enter2] :
{ (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId’]: VList) | VSet ;; (v[
    VId]: LId, crossing, VT, VT) ;; (v[VId’]: LId’, stopped, VT2, VT2) }
=>
if isPriorityLane(LId) and isPriorityLane(LId’) and not areConflict(LId, LId’)
then
    { (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId’]: VList) | (v[VId]:
        LId, crossing, VT, VT) ;; (v[VId’]: LId’, crossing, VT2, VT2) ;; letCross(VList, VSet)
         }
else
    { (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId’]: VList) | VSet ;; (
        v[VId]: LId, crossing, VT, VT) ;; (v[VId’]: LId’, waiting, VT2, VT2) }
fi [narrowing] .
```

Exit

Lastly, the exit rule allows the vehicle to end its interaction with the intersection by exiting it, freeing the it for the next vehicle or set of vehicles.

This rule first checks if all vehicles that were circulating have approached the intersection and if the intersection itself is occupied. It also checks if there are vehicles in the *waiting* state once the current vehicle has crossed.

```
--- Exit: it makes the vehicle go from CROSSING to CROSSED. Exit intersection
--- One transition:
rl [exit1] :
{ (fstat: nEnd) | (istat: true) | (clock: T, B, B’) | LSet ;; (l[LId]: (VId VList)) | VSet ;;
    (v[VId]: LId, crossing, VT, VT’) }
=>
if haveApproached?(VSet)
then
    if isOccupied(VSet)
    then
        { (fstat: nEnd) | (istat: true) | (clock: T, B, true) | LSet ;; (l[LId]: (VList)) | (v
            [VId]: LId, crossed, VT, VT’) ;; checkWaiting(VSet) }
    else
```

```
        { (fstat: nEnd) | (istat: false) | (clock: T, B, true) | LSet ;; (l[LId]: (VList)) | (
            v[VId]: LId, crossed, VT, VT') ;; checkWaiting(VSet) }
    fi
else
    if isOccupied(VSet)
    then
        { (fstat: nEnd) | (istat: true) | (clock: T, B, B') | LSet ;; (l[LId]: (VList)) | (v[
            VId]: LId, crossed, VT, VT') ;; checkWaiting(VSet) }
    else
        { (fstat: nEnd) | (istat: false) | (clock: T, B, B') | LSet ;; (l[LId]: (VList)) | (v[
            VId]: LId, crossed, VT, VT') ;; checkWaiting(VSet) }
    fi
fi [narrowing] .
```

The auxiliary equations `haveApproached?`, `isOccupied` and `checkWaiting` serve the purpose of seeing if the execution can end because there are no more new approaching vehicles, if the intersection should still be marked as occupied or not and to change the state of all waiting vehicles to *stopped* so as to continue the execution, respectively.

```
--- Check if all vehicles have approached the intersection
op haveApproached? : VehicleSet -> Bool .
eq haveApproached?(none) = true .
eq haveApproached?((v[VId]: LId, VSt, VT, VT') ;; VSet) =
    VSt =/= circulating and-then end?(VSet) [variant] .


op isOccupied : VehicleSet -> Bool .
eq isOccupied(none) = false .
eq isOccupied((v[VId]: LId, VSt, VT, VT') ;; VSet) =
    if VSt == crossing
    then true
    else isOccupied(VSet)
    fi [variant] .


--- Set all waiting cars from WAITING to STOPPED again
op checkWaiting : VehicleSet -> VehicleSet .
eq checkWaiting(none) = none .
eq checkWaiting(((v[VId]: LId, VSt, VT, VT') ;; VSet)) =
    if VSt == waiting
    then
        (v[VId]: LId, stopped, VT, VT') ;; checkWaiting(VSet)
    else
        (v[VId]: LId, VSt, VT, VT') ;; checkWaiting(VSet)
    fi [variant] .
```

Reset, Clock and End

Additionally, three more rules exist to control the advancement of time, resetting the head vehicle of a lane as well as checking if the execution has ended.

As we can see here, the `tick` rule can only advance time once a vehicle has arrived:

```
--- Tick: ticking of clock to pass time. If all vehicles have approached the intersection, the
     execution can continue until properly ending.
rl [tick] :
{ (fstat: nEnd) | (istat: false) | (clock: T, true, B) | LSet | VSet }
=>
{ (fstat: nEnd) | (istat: false) | (clock: (T + 1) rem 10, false, B) | LSet | VSet } [
    narrowing] .
```

This rule is not only useful to track the execution, but serves us also as a tool to verify certain properties, like not being able to have two vehicles with the same timestamps on different lanes at the same time.

Meanwhile, `reset` covers any scenario in which the head vehicle has crossed the intersection and a vehicle that was too far away and could not cross is now left *stopped* at the intersection. This rule turns the first vehicle that is stopped in that lane into the new head of the lane, allowing it and any others that are behind to cross the intersection properly. This rule uses the auxiliary equation `resetHead`:

```
--- Reset: when head vehicle has crossed the intersection, make new first stopped vehicle the
    head of the lane.
--- One transition:
--- Rule for head vehicle stopping
rl [reset] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, B) | LSet ;; (l[LId]: VId’ VList) | VSet
    ;; (v[VId]: LId, crossed, VT, VT) ;; (v[VId’]: LId, stopped, VT, VT’) }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, B) | LSet ;; (l[LId]: VId’ VList) | (v[
    VId]: LId, crossed, VT, VT) ;; (v[VId’]: LId, stopped, VT’, VT’) ;; resetHead(VId’ VList,
    (v[VId’]: LId, stopped, VT’, VT’), VSet) }  [narrowing] .


op resetHead : NatList Vehicle VehicleSet -> VehicleSet .
eq resetHead(VId VList, (v[VId]: LId, stopped, VT, VT’), none) = none [variant] .
eq resetHead(VId VList, (v[VId]: LId, stopped, VT, VT’), (v[VId’]: LId, VSt, VT2, VT2’) ;;
    VSet) =
    if VId’ in VList and VSt == stopped
    then (v[VId’]: LId, stopped, VT’, VT2’) ;; resetHead(VId VList, (v[VId]: LId, stopped, VT,
         VT’), VSet)
    else (v[VId’]: LId, VSt, VT2, VT2’) ;; resetHead(VId VList, (v[VId]: LId, stopped, VT, VT
        ’), VSet)
    fi [variant] .
```

Lastly, end is a rule that uses the equation `end?` to make sure all vehicles in the state have crossed. Here we can see both the equation and the corresponding rule:

```
--- Check if all vehicles have crossed the intersection
op end? : VehicleSet -> Bool .
eq end?(none) = true .
eq end?((v[VId]: LId, VSt, VT, VT’) ;; VSet) =
    VSt == crossed and-then end?(VSet) [variant] .


rl [end] :
{ (fstat: nEnd) | (istat: false) | (clock: VT, B, true) | (l[0]: nil) ;; (l[1]: nil) ;; (l[2]:
     nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | VSet } =>
if end?(VSet)
then
    { (fstat: end) | (istat: false) | (clock: VT, B, true) | (l[0]: nil) ;; (l[1]: nil) ;; (l
        [2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | VSet }
else
    { (fstat: nEnd) | (istat: false) | (clock: VT, B, true) | (l[0]: nil) ;; (l[1]: nil) ;; (l
        [2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | VSet }
fi [narrowing] .
```

During the process of creation of this model, the `Anima` tool [3] was used, as it helped visualize how the model behaved and it helped with early stages of error-finding. It was also used in the model-checking stages of the project as it helped to better visualize the `counterexamples` provided by the model-checker.

Chapter 4

Analysis and Verification of the AV Model

In this Chapter, we proceed to explain the different techniques that we use to verify our model: `Maude's` built-in classical LTL model-checker and `Maude`'s recently optimized *narrowing* infrastructure.

4.1 Classical model-checking analysis

After having developed our model specification as a rewrite theory and implemented all of the `INTERSECTION` modules, we can now proceed to the verification. This is done through the composition of *atomic propositions (AP)*, which are predicates referring to the different states of the system that can be fulfilled, and LTL formulas.

Models of temporal logic can be understood as Kripke structures, which are important to define predicates about said model. A Kripke structure is correlatable to a rewrite theory specified in a Maude system module `M`.

A Kripke structure can be seen as a transition system in which each of its nodes represents a reachable state and is generally used in model-cheking to represent the behavior of a system. It can be defined as a 4-tuple $\mathcal{M} = (S, S_0, R, L)$ where S is the finite set of states, S_0 is the initial state, R is the transition relation between states and L is the labeling function $L : S \to 2^{AP}$.

We associate a Kripke structure to a rewrite theory specified in `Maude` by expounding both the intended *kind k* of states in the signature Σ and the relevant set of *atomic propositions*, which conform the state predicates Π.

In order to achieve this, we first define a module INTERSECTION-PREDS, which imports the model-checker defined in the module SATISFACTION. Within this module, we create a subsort association between our type of state Config and the type State from the SATISFACTION module. This Config will represent the states in our system as State in the Kripke structure. We then create the *atomic propositions*: executionEnd, inIntersection and waitIntersection [1].

```
*** Configuration as subsort of State ***
subsort Config < State .

*** Declaration of predicates ***
--- All vehicles have crossed the intersection successfully
op executionEnd : -> Prop .
--- A vehicle with Id (Nat) is occupying the intersection
op inIntersection : Nat -> Prop .
op waitIntersection : Nat -> Prop .
```

After this, we can define our equations, which follow the structure
<State> $\models$ <Predicate>.

The first equation's purpose is to define when the execution has ended successfully, meaning all vehicles have crossed the intersection. The second and third equations define when a vehicle is occupying the intersection and when it is waiting to enter, respectively. These equations allow us to verify mutual exclusion and liveness properties. Subsequently we have the equations as:

```
*** Equations for state predicates ***
--- Equation for when reaching the end state fulfilled. All vehicles have crossed the
    intersection successfully
eq { (fstat: end) | (istat: false) | (clock: T, false, true) | (l[0]: nil) ;; (l[1]: nil) ;; (
    l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | VSet } |= executionEnd = true .

--- Equation that checks the intersection's state and the state of the vehicle (occupied)
eq { (fstat: nEnd) | (istat: true) | (clock: T, B, B') | LSet | VSet ;; (v[VId]: LId, crossing
    , VT, VT') } |= inIntersection(VId) = true .

--- Equation that checks the intersection's state and the state of the vehicle (waiting)
eq { (fstat: nEnd) | (istat: B) | (clock: T, B', B'') | LSet | VSet ;; (v[VId]: LId, stopped,
    VT, VT') } |= waitIntersection(VId) = true .
```

Now that we have the correlation between our rewrite theory and the Kripke structure, we can define an initial state in which we can prove our *atomic properties*. The assortment of reachable states defined by the initial state must be finite and the equations D that define the predicates Π in addition to the rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$, defined in in INTERSECTION-PREDS, are such that both E and $E \cup D$ are Church-Rosser [2] and terminating and R is ground [3].

[1]Although in LTL predicates are constants, Maude allows the definition of predicates with parameters, enabling the possibility of checking a property for a specific parameter such as the ID of a vehicle, for example.

[2]The order of application of equations does not affect the obtained result.

[3]The rules of the system do not contain variables.

Assuming the compliance of these conditions, we define the module INTERSECTION-CHECK, which imports INTERSECTION-PREDS as well as MODEL-CHECK and the optional model LTL-SIMPLIFIER, which is used for efficiency purposes, simplifying the negative normal form of a formula.

In the following code section we declare the initial states that represent different encounters at an intersection that can provide interesting scenarios to be verified. These initial states are defined as operators with their corresponding equations: {init1, ..., init5}. All initial states have a finality status of *not ended*, an intersection status of *empty*, a clock that starts at 0 in which time cannot advance nor can the execution end as well as a set of empty lanes, leaving the variation between these states to be the set of vehicles that will affect the execution:

- init1: Two vehicles in parallel, priority, lanes are circulating.
- init2: Two vehicles in conflicting, one priority and another secondary, lanes are circulating.
- init3: Two vehicles in conflicting, priority, lanes are circulating.
- init4: Four vehicles in the same lane are circulating.
- init5: Four vehicles from several lanes, priority and secondary as well as parallel and conflicting, are circulating.

```
*** Initial States ***
ops init1 init2 init3 init4 init5 : -> Config .

--- Definition of initial state
eq init1 = {
        (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
        (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
        (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
        (v[0]: 0, circulating, 100, 100) ;;
        (v[1]: 1, circulating, 100, 100)
    } .

eq init2 = {
        (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
        (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
        (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
        (v[0]: 0, circulating, 100, 100) ;;
        (v[1]: 2, circulating, 100, 100)
    } .

eq init3 = {
        (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
        (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
        (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
        (v[0]: 0, circulating, 100, 100) ;;
        (v[1]: 4, circulating, 100, 100)
    } .

eq init4 = {
        (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
        (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
        (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
        (v[0]: 0, circulating, 100, 100) ;;
        (v[1]: 0, circulating, 100, 100) ;;
        (v[2]: 0, circulating, 100, 100) ;;
        (v[3]: 0, circulating, 100, 100)
    } .

eq init5 = {
        (fstat: nEnd) | (istat: false) | (clock: 0, false, false) |
        (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;;
        (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) |
        (v[0]: 0, circulating, 100, 100) ;;
        (v[1]: 2, circulating, 100, 100) ;;
        (v[2]: 3, circulating, 100, 100) ;;
        (v[3]: 4, circulating, 100, 100)
    } .
```

Mutual exclusion

In order to verify mutual exclusion, we will consider the initial states 1, 2 and 3, which represent interesting scenarios for proving mutual exclusion, barring the fact that parallel lanes do not provided for mutual exclusion as their feature is allowing parallelism.

For the property of mutual exclusion, we must guarantee that vehicles that circulate across conflicting lanes will not be able to access the critical section, the intersection, at the same time. Formulated in natural language, the property we desire is expressed as:

For all possible future states, it is always true that two conflicting vehicles will not be in the intersection at the same time.

Therefore, the LTL formula we would be interested in proving would be:
$G\neg(inIntersection(0) \wedge inIntersection(1))$.

For the states `init2` and `init3` we obtain the following when running the model checker, proving mutual exclusion is guaranteed:

```
Maude> red modelCheck(init2, []~ (inIntersection(0) /\ inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init2, []~ (inIntersection(0) /\ inIntersection(1)))
    .
rewrites: 1013 in 3ms cpu (4ms real) (280454 rewrites/second)
result Bool: true
```

```
Maude> red modelCheck(init3, []~ (inIntersection(0) /\ inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init3, []~ (inIntersection(0) /\ inIntersection(1)))
    .
rewrites: 1273 in 2ms cpu (5ms real) (456435 rewrites/second)
result Bool: true
```

The same cannot be said for `init1`, as the vehicles in question reside in parallel lanes that allows concurrency within the intersection. Therefore, when running the model-checker, we are provided with a *counterexample* (given in page 53 of the Appendix tagged as `1)`) that proves mutual exclusion is not possible, just as intended.

Strong liveness

Strong liveness verification can be proven by considering the initial states 1, 2 and 3 again, as we want to check that no conflicting or parallel vehicles are waiting infinitely.

For the property of strong liveness, we must guarantee that all vehicles that are waiting will, eventually, be able to access the critical section. Formulated in natural language, the property we desire is expressed as:

For all possible future states, it is always true that if a process waits infinitely often, then it is in its critical section infinitely often.

Therefore, the LTL formula we would be interested in proving would be:
$GF(waitIntersection(0) \rightarrow inIntersection(0))$.

For the state `init1`, obtain the following when running the model checker:

```
Maude> red modelCheck(init1, ([] <> waitIntersection(0)) -> ([] <> inIntersection(0))) .
reduce in INTERSECTION-CHECK : modelCheck(init1, []<> waitIntersection(0) -> []<>
     inIntersection(0)) .
rewrites: 1339 in 4ms cpu (5ms real) (273992 rewrites/second)
result Bool: true
```

```
Maude> red modelCheck(init1, ([] <> waitIntersection(1)) -> ([] <> inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init1, []<> waitIntersection(1) -> []<>
     inIntersection(1)) .
rewrites: 1339 in 3ms cpu (3ms real) (362676 rewrites/second)
result Bool: true
```

Similar results are provided for the states `init2` and `init3` in page 54 of the Appendix (tagged as `2)` and `3)`), proving strong liveness is guaranteed.

End of execution

End of execution verification can be proven by considering the initial states 4 and 5 (initial states 1, 2 and 3 can also be used, but are considered trivial), as these states include several vehicles that provide a more interesting configuration that could lead to deadlocks or starvation.

This last property seeks to demonstrate that all vehicles in the execution will eventually cross, proving there are no deadlocks and the system is starvation free. Formulated in natural language, the property we desire is expressed as:

For all possible future states, it is always true that the execution will eventually reach the state flagged as `end`, indicating all vehicles have crossed the intersection.

Therefore, the LTL formula we would be interested in proving would be: $GFexecutionEnd$.

Before continuing to the results, it is interesting to note that the above property didn't hold in the first version of our model and finding this error led us to correct our original specification. Actually, it would be remiss to not explore the error in the model that was found through model-checking.

Intuitively, the `stop` rule could be written as four separate rules: one rule for when the head vehicle wants to stop and three for when a non-head vehicle wants to stop and the head vehicle has already stopped at some point. While this set of rules seemed to work as intended, they did not control order of arrival when stopping at the intersection. This meant that a vehicle farther back in the list of a given lane could stop before one that had not stopped yet, generating an inconsistency that would result in a deadlock when asking the model-checker to verify the property. `Maude`'s model-checker would return a *counterexample* that can be also found in page 55 of the Appendix tagged as `4)`.

These were the rules first used in our model that generated the *counterexample* in question:

```
rl [stop1] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    stopping, VT, VT) }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    stopped, VT, VT) } [narrowing] .

rl [stop2] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    stopped, VT, VT) ;; (v[VId]: LId, stopping, VT, VT') }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    stopped, VT, VT) ;; (v[VId]: LId, stopped, VT, VT') } [narrowing] .

rl [stop3] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    crossing, VT, VT) ;; (v[VId]: LId, stopping, VT, VT') }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    crossing, VT, VT) ;; (v[VId]: LId, stopped, VT, VT') } [narrowing] .

rl [stop4] :
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    crossed, VT, VT) ;; (v[VId]: LId, stopping, VT, VT') }
=>
{ (fstat: nEnd) | (istat: false) | (clock: T, false, false) | LSet | VSet ;; (v[VId]: LId,
    crossed, VT, VT) ;; (v[VId]: LId, stopped, VT, VT') } [narrowing] .
```

Once corrected, we can see that for the same initial state `init5`, we now have a successful result, as order of arrival is controlled in the current model:

```
Maude> red modelCheck(init5, [] <> executionEnd) .
reduce in INTERSECTION-CHECK : modelCheck(init5, []<> executionEnd) .
rewrites: 493527 in 214ms cpu (224ms real) (2302574 rewrites/second)
result Bool: true
```

End of execution is verified as well when providing several vehicles from different lanes, thus ensuring correct behavior:

```
Maude> red modelCheck(init4, [] <> executionEnd) .
reduce in INTERSECTION-CHECK : modelCheck(init4, []<> executionEnd) .
rewrites: 184996 in 144ms cpu (155ms real) (1280027 rewrites/second)
result Bool: true
```

4.2 Symbolic, narrowing-based analysis

Once we have finished with verifying our model through model-checking, we can proceed to applying *narrowing* to our model through `Maude`'s built in *narrowing* infrastructure. This allows us to deal with systems where 1) the number of initial states can be very big or even infinite (for example, systems parametric in the number of processes or objects); or 2) we want to perform an analysis that is independent of said parameters. This second way of applying *narrowing* allows us to see all possible solutions to a query, which we can then analyze to ensure no unwanted solutions are provided. In order to achieve this, we must ignore the `INTERSECTION-PREDS` and `INTERSECTION-CHECK` modules, as they cannot be used for *narrowing*. We must also ensure we have added `[variant]` to all equations and `[narrowing]` to all rules in which *narrowing* is applied.

For *narrowing* we use the `vu-narrow` command as well as the `Narval` tool [2] for visualization when applying *narrowing*, similarly to how we made use of the `Anima` tool in the implementation and model-checking sections of the project.

Maintaining order

The first property we want to ensure through *narrowing* would be the fact that vehicles maintain order, meaning a vehicle that is behind another cannot stop at the intersection before this one. With the initial state in which we have two vehicles

```
{ (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil) ;; (l[1]: nil) ;; (l
    [2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0, circulating, 100, 100)
    ;; (v[1]: X:Nat, circulating, 100, 100) } .
```

and the objective state, in which the vehicle with `ID [1]` has stopped before the vehicle with `ID [0]`

```
{ (fstat: nEnd) | (istat: false) | (clock: C:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l
    [1]: NL1:NatList) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;;
    (l[5]: NL5:NatList) | (v[0]: 0, stopping, 0, 0) ;; (v[1]: X:Nat, stopped, T:Nat, T':Nat) }
     .
```

We use the vehicle with `ID [0]` as our reference vehicle. It arrives before vehicle `ID [1]` at the intersection but is still in the process of decelerating, meaning it has not arrived at the `stopped` state yet. Meanwhile vehicle `ID [1]` has already `stopped` before `ID [0]` although it arrived afterwards. We choose the `ID` of the Lane as the free variable, as we want to check what lanes could possibly allow for a vehicle with a succeeding timestamp to stop before a vehicle that has arrived earlier. As we checked previously, in Section 4.1, vehicles belonging to the same lane are unable to stop before the one that is in front of them. With this free variable we are able to check if this is true for all lanes, meaning all vehicles arrive in order at the intersection, before stopping. When we check the reachability of the state, we obtain the expected result in which all vehicles must stop in order of arrival and therefore finding no solution to our query:

```
Maude> vu-narrow { (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil) ;;
    (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0,
    circulating, 100, 100) ;; (v[1]: X:Nat, circulating, 100, 100) } =>* { (fstat: nEnd) | (
    istat: false) | (clock: C:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l[1]: NL1:
    NatList) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;; (l[5]:
    NL5:NatList) | (v[0]: 0, stopping, 0, 0) ;; (v[1]: X:Nat, stopped, T:Nat, T':Nat) } .
vu-narrow in INTERSECTION-RL : {fstat: nEnd | istat: false | clock: 0,false,false | ((((l[4]:
     nil) ;; l[5]: nil) ;; l[3]: nil) ;; l[2]:
    nil) ;; l[1]: nil) ;; l[0]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: X:Nat,circulating
        ,100,100} =>* {fstat: nEnd | istat: false |
    clock: C:Nat,B,B' | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList) ;; (l[2]: NL2:NatList) ;; (
        l[3]: NL3:NatList) ;; (l[4]: NL4:NatList)
    ;; l[5]: NL5:NatList | (v[0]: 0,stopping,0,0) ;; v[1]: X:Nat,stopped,T:Nat,T':Nat} .

No solution.
rewrites: 21511 in 6026ms cpu (7606ms real) (3569 rewrites/second)
```

Maintaining lane priority

Just as we would want to check for the fact that order is maintained, we should also check the fact that lane priority is maintained. As stated in Section 4.1, vehicles with lower priority cannot cross when a vehicle of higher priority is in the intersection, but it should also be stated that they should not be able to cross even if the intersection is unoccupied if there is a priority vehicle waiting to enter, as they are required to wait when this occurs. Therefore, with the initial state

```
{ (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil) ;; (l[1]: nil) ;; (l
    [2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0, circulating, 100, 100)
    ;; (v[1]: X:Nat, circulating, 100, 100) } .
```

and the target state

```
{ (fstat: nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l
    [1]: NL1:NatList) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;;
    (l[5]: NL5:NatList) | (v[0]: 0, stopped, T1:Nat, T1':Nat) ;; (v[1]: X:Nat, crossing, T2:
    Nat, T2':Nat) } .
```

in which we have a reference (priority) vehicle with `ID [0]` in Lane 0 that has stopped and we would want to find out what characteristics the vehicle with `ID [1]` should have in order to cross before vehicle `ID [0]`. Our variable targets the Lane vehicle `ID [1]` belongs to. We should expect the solution substitution for said Lane's `ID` to be any valid `ID` other than `ID [2]` or `ID [5]`, meaning secondary vehicles enter a state of waiting and cannot cross before a priority vehicle if it is nearby. Therefore, we obtain the following solution [4] in which we obtain the lane a vehicle should be

[4]We constrained this solution to the first result so as to save space in this section. For the

occupying in order to enter the intersection without respecting the lane priority, in this case Lane 1 [5]:

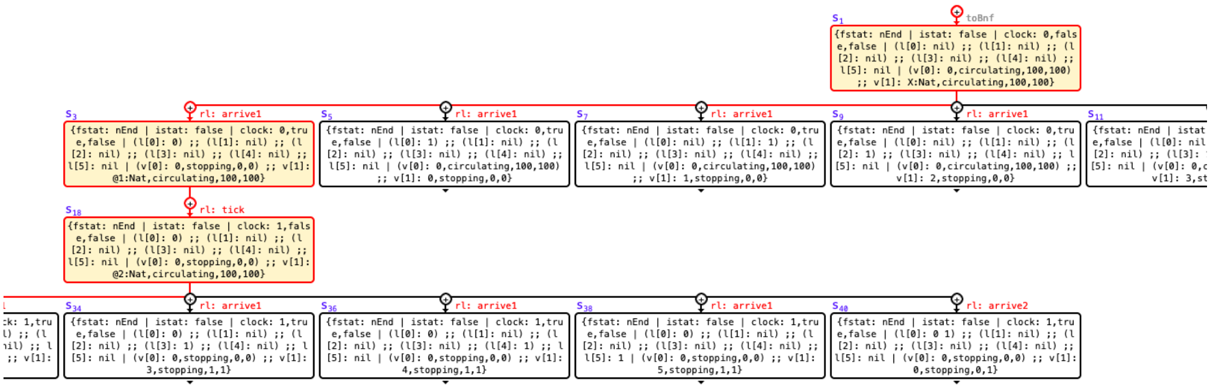
```
Maude> vu-narrow [1] { (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil)
     ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0,
    circulating, 100, 100) ;; (v[1]: X:Nat, circulating, 100, 100) } =>* { (fstat: nEnd) | (
    istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList
    ) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;; (l[5]: NL5:
    NatList) | (v[0]: 0, stopped, T1:Nat, T1':Nat) ;; (v[1]: X:Nat, crossing, T2:Nat, T2':Nat)
     } .
vu-narrow [1] in INTERSECTION-RL : {fstat: nEnd | istat: false | clock: 0,false,false | (((((l
    [4]: nil) ;; l[5]: nil) ;; l[3]: nil) ;; l[
    2]: nil) ;; l[1]: nil) ;; l[0]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: X:Nat,
        circulating,100,100} =>* {fstat: nEnd | istat:
    true | clock: T:Nat,B,B' | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList) ;; (l[2]: NL2:
        NatList) ;; (l[3]: NL3:NatList) ;; (l[4]:
    NL4:NatList) ;; l[5]: NL5:NatList | (v[0]: 0,stopped,T1:Nat,T1':Nat) ;; v[1]: X:Nat,
        crossing,T2:Nat,T2':Nat} .

Solution 1
rewrites: 2583 in 660ms cpu (767ms real) (3911 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,crossing,1,1}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1
```

We can also visualize this solution state thanks to the `Narval` tool, which provides us with a symbolic execution tree. If we specify a target state, it will be highlighted as a green node, meaning the desired state was indeed reached. As we can see in Figure 4.1 and Figure 4.2, `Narval` provides the same solution we have shown before, in which the Lane's `ID` is `[1]` and vehicle `ID [0]` arrived before vehicle `ID [1]`.

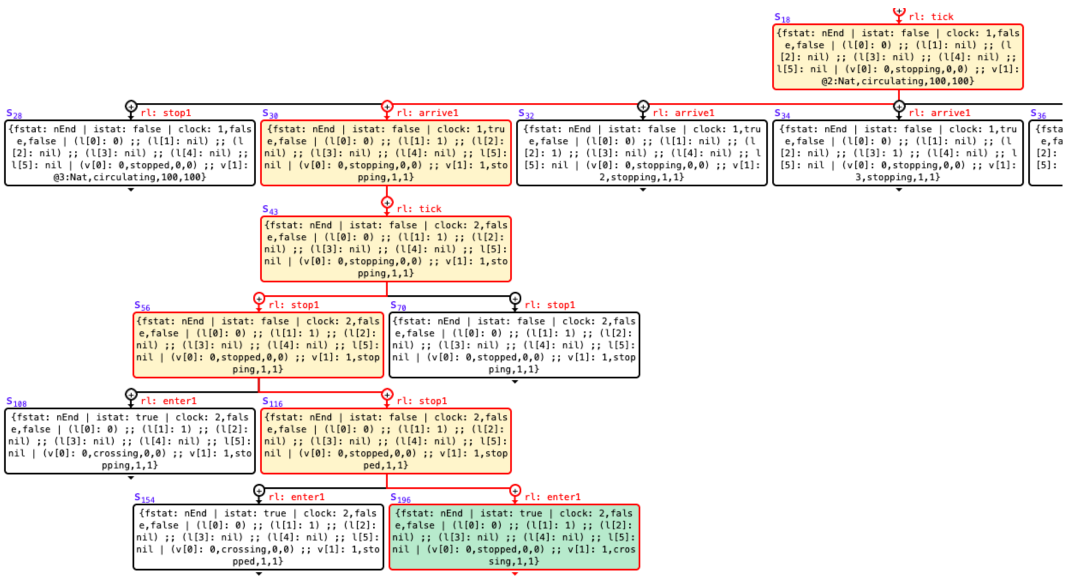
results with no constraints check page 61 of the the Appendix (tagged as `5)`).

[5]In the aforementioned unconstrained solution provided in the Appendix we can see that the `IDs` of the Lanes are 0, 1, 3 and 4, as these are the lanes with the same level of priority as the Lane 0. As we proved in the preceding property (*Maintaining order*), the solution vehicle in Lane 0 would have an earlier timestamp than the reference vehicle, thus maintaining order.



s_1
toBnf
{fstat: nEnd | istat: false | clock: 0,false,false | (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: X:Nat,circulating,100,100}
s_3
rl: arrive1
{fstat: nEnd | istat: false | clock: 0,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: @1:Nat,circulating,100,100}
s_5
rl: arrive1
{fstat: nEnd | istat: false | clock: 0,true,false | (l[0]: 1) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: 0,stopping,0,0}
s_7
rl: arrive1
{fstat: nEnd | istat: false | clock: 0,true,false | (l[0]: nil) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: 1,stopping,0,0}
s_9
rl: arrive1
{fstat: nEnd | istat: false | clock: 0,true,false | (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: 2,stopping,0,0}
s_{11}
s_{18}
rl: tick
{fstat: nEnd | istat: false | clock: 1,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: @2:Nat,circulating,100,100}
s_{34}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 3,stopping,1,1}
s_{36}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 4,stopping,1,1}
s_{38}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: 1 | (v[0]: 0,stopping,0,0) ;; v[1]: 5,stopping,1,1}
s_{40}
rl: arrive2
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0 1) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 0,stopping,0,1}

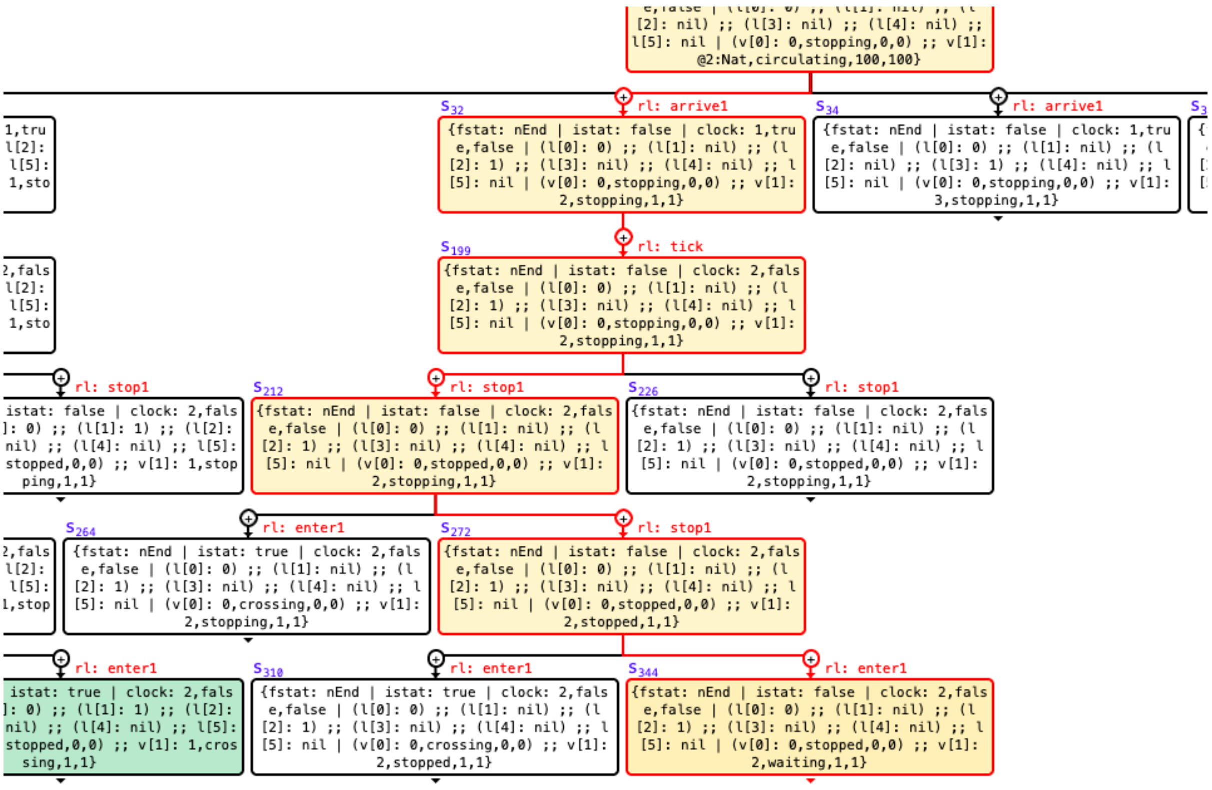
Figure 4.1: Initial state and start of the execution path



s_{18}
rl: tick
{fstat: nEnd | istat: false | clock: 1,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: @2:Nat,circulating,100,100}
s_{28}
rl: stop1
{fstat: nEnd | istat: false | clock: 1,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: @3:Nat,circulating,100,100}
s_{30}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 1,stopping,1,1}
s_{32}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 2,stopping,1,1}
s_{34}
rl: arrive1
{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 3,stopping,1,1}
s_{36}
s_{43}
rl: tick
{fstat: nEnd | istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 1,stopping,1,1}
s_{56}
rl: stop1
{fstat: nEnd | istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,stopping,1,1}
s_{70}
rl: stop1
{fstat: nEnd | istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,stopping,1,1}
s_{108}
rl: enter1
{fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,crossing,0,0) ;; v[1]: 1,stopping,1,1}
s_{116}
rl: stop1
{fstat: nEnd | istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,stopped,1,1}
s_{154}
rl: enter1
{fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,crossing,0,0) ;; v[1]: 1,stopped,1,1}
s_{196}
rl: enter1
{fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,crossing,1,1}

Figure 4.2: Path to successful arrival at target state

On the other hand, with the same initial state and the same vehicle `ID [1]` characteristics, when we assign vehicle `ID [1]` to Lane 2, it will wait, as it is not allowed to cross before vehicle `Id [0]`, as we can see in Figure 4.3.



```
[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;;
l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]:
@2:Nat,circulating,100,100}

S32
{fstat: nEnd | istat: false | clock: 1,tru
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]:
2,stopping,1,1}

S34
{fstat: nEnd | istat: false | clock: 1,tru
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]:
3,stopping,1,1}

S199
{fstat: nEnd | istat: false | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]:
2,stopping,1,1}

istat: false | clock: 2,fals
]: 0) ;; (l[1]: 1) ;; (l[2]:
nil) ;; (l[4]: nil) ;; l[5]:
stopped,0,0) ;; v[1]: 1,stop
ping,1,1}

S212
{fstat: nEnd | istat: false | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]:
2,stopping,1,1}

S226
{fstat: nEnd | istat: false | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]:
2,stopping,1,1}

S264
{fstat: nEnd | istat: true | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,crossing,0,0) ;; v[1]:
2,stopping,1,1}

S272
{fstat: nEnd | istat: false | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]:
2,stopped,1,1}

istat: true | clock: 2,fals
]: 0) ;; (l[1]: 1) ;; (l[2]:
nil) ;; (l[4]: nil) ;; l[5]:
stopped,0,0) ;; v[1]: 1,cros
sing,1,1}

S310
{fstat: nEnd | istat: true | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,crossing,0,0) ;; v[1]:
2,stopped,1,1}

S344
{fstat: nEnd | istat: false | clock: 2,fals
e,false | (l[0]: 0) ;; (l[1]: nil) ;; (l
[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]:
2,waiting,1,1}
```

Figure 4.3: Path to unsuccessful arrival at target state

Vehicle advancement

Lastly, we should want to check that vehicles cannot backtrack during the execution, meaning a vehicle with `crossing` status cannot go back into a `stopped` status and similar variations. This ensures there are no loops in the model and that vehicles cannot reach states that allow them to cross the intersection multiple times or execute other abnormal behaviors.

Consider the initial state

```
{ (fstat: nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: 0 N0:NatList) ;; (l
    [1]: N1:NatList) ;; (l[2]: N2:NatList) ;; (l[3]: N3:NatList) ;; (l[4]: N4:NatList) ;; (l
    [5]: N5:NatList) | (v[0]: X:Nat, crossing, VT:Nat, VT':Nat) } .
```

and the target state

```
{ (fstat: nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: 0 N0:NatList) ;; (l
    [1]: N1:NatList) ;; (l[2]: N2:NatList) ;; (l[3]: N3:NatList) ;; (l[4]: N4:NatList) ;; (l
    [5]: N5:NatList) | (v[0]: X:Nat, stopped, VT:Nat, VT':Nat) } .
```

We note that there are no possible configurations for which this should be true. Vehicle lane, time of arrival, state of the clock or lane contents should not influence the vehicle's behavior in such a way that it would arrive to a state it has already been in. We can see that no solutions can be provided, as no loops that would allow this behavior exist, thus ensuring the correct behavior of the model:

```
Maude> vu-narrow { (fstat: nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: 0
    N0:NatList) ;; (l[1]: N1:NatList) ;; (l[2]: N2:NatList) ;; (l[3]: N3:NatList) ;; (l[4]: N4
    :NatList) ;; (l[5]: N5:NatList) | (v[0]: X:Nat, crossing, VT:Nat, VT':Nat) } =>* { (fstat:
     nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: 0 N0:NatList) ;; (l[1]:
    N1:NatList) ;; (l[2]: N2:NatList) ;; (l[3]: N3:NatList) ;; (l[4]: N4:NatList) ;; (l[5]: N5
    :NatList) | (v[0]: X:Nat, stopped, VT:Nat, VT':Nat) } .
vu-narrow in INTERSECTION-RL : {fstat: nEnd | istat: true | clock: T:Nat,B,B' | (((((l[4]: N4:
    NatList) ;; l[5]: N5:NatList) ;; l[3]:
    N3:NatList) ;; l[2]: N2:NatList) ;; l[1]: N1:NatList) ;; l[0]: 0 N0:NatList | v[0]: X:Nat,
        crossing,VT:Nat,VT':Nat} =>* {fstat: nEnd |
    istat: true | clock: T:Nat,B,B' | (l[0]: 0 N0:NatList) ;; (l[1]: N1:NatList) ;; (l[2]: N2:
        NatList) ;; (l[3]: N3:NatList) ;; (l[4]:
    N4:NatList) ;; l[5]: N5:NatList | v[0]: X:Nat,stopped,VT:Nat,VT':Nat} .

No solution.
rewrites: 54 in 110ms cpu (169ms real) (489 rewrites/second)
```

Chapter 5

Conclusions and Future Work

Behavioral protocols and their implementations for autonomous vehicles are in high demand. This new and rapidly growing technology requires the study of an immense amount of scenarios, which themselves require standardization and verification.

This MSc Thesis proposes a verifiable variation of the *LPJL* protocol. We have been able to define a formal specification and model of said variant protocol in the `Maude` language, which was subsequently thoroughly verified through both `Maude`'s model-checker and its *narrowing* built-in infrastructure. We were also able to enhance both the creation of the model and its following analysis thanks to the `Anima` and `Narval` tools, which provided visual aid by representing the execution of the model through a tree whose nodes are each reachable state of the execution obtained through applying the equations and rules declared in the model.

As a future addition to this work, more scenarios could be considered to be verified such as platooning (a method for driving a group of vehicles together), incorporation into a lane, or overtaking another vehicle. These scenarios could be modeled to allow for a *narrowing* analysis to be made, as this approach is not widely used and could aid with efficiency. Another future addition would be to make use of `Maude`'s symbolic model-checker. This model-checker was not used as it both requires the use of `Full Maude`, it is also rather inefficient, and it has not been updated for the version of `Maude` that was used for this project.

Bibliography

[1] S. Agrawal and B. Bonakdarpour. “Runtime Verification of k-Safety Hyperproperties in HyperLTL”. In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 239–252. URL: `https://doi.org/10.1109/CSF.2016.24`.

[2] M. Alpuente, D. Ballis, S. Escobar, and J. Sapiña. “Symbolic Analysis of Maude Theories with Narval”. In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 874–890. URL: `https://doi.org/10.1017/S1471068419000243`.

[3] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. “Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way)”. In: *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*. Ed. by S. Iida, J. Meseguer, and K. Ogata. Vol. 8373. Lecture Notes in Computer Science. Springer, 2014, pp. 229–255. URL: `https://doi.org/10.1007/978-3-642-54624-2%5C_12`.

[4] S. Aoki and R. Rajkumar. “A merging protocol for self-driving vehicles”. In: *Proceedings of the 8th International Conference on Cyber-Physical Systems, ICCPS 2017, Pittsburgh, Pennsylvania, USA, April 18-20, 2017*. Ed. by S. Martinez, E. Tovar, C. Gill, and B. Sinopoli. ACM, 2017, pp. 219–228. URL: `https://doi.org/10.1145/3055004.3055028`.

[5] J. Arcile, R. R. Devillers, and H. Klaudel. “VerifCar: a framework for modeling and model checking communicating autonomous vehicles”. In: *Auton. Agents Multi Agent Syst.* 33.3 (2019), pp. 353–381. URL: `https://doi.org/10.1007/s10458-019-09409-x`.

[6] M. N. Aung, Y. Phyo, and K. Ogata. “Formal Specification and Model Checking of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol (S)”. In: *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*. Ed. by A. Perkusich. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019, pp. 159–208. URL: `https://doi.org/10.18293/SEKE2019-021`.

[7] Autocrypt. *DSRC vs. C-V2X: A Detailed Comparison of the 2 Types of V2X Technologies*. URL: `https://autocrypt.io/dsrc-vs-c-v2x-a-detailed-comparison-of-the-2-types-of-v2x-technologies/`.

[8] Society of Automotive Engineers. *Introduction to Highly Automated Vehicles*. URL: `https://www.sae.org/learn/content/c1603/`.

[9] S. Azimi, G. Bhatia, R. Rajkumar, and P. Mudalige. "STIP: Spatio-temporal intersection protocols for autonomous vehicles". In: *ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS, Berlin, Germany, April 14-17, 2014*. IEEE Computer Society, 2014, pp. 1–12. URL: `https://doi.org/10.1109/ICCPS.2014.6843706`.

[10] C. Baier and J. P. Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[11] R. Butler. *Langley Formal Methods Program. What is Formal Methods*. URL: `https://shemesh.larc.nasa.gov/fm/fm-what.html`.

[12] E. M. Clarke, E. A. Emerson, and J. Sifakis. "Model checking: algorithmic verification and debugging". In: *Commun. ACM* 52.11 (2009), pp. 74–84. URL: `https://doi.org/10.1145/1592761.1592781`.

[13] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Marti-Oliet, J. Messeguer, and C. Talcott. *Maude Manual (Version 3.2.1)*. URL: `https://maude.lcc.uma.es/maude321-manual-html/maude-manual.html`.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. L. Talcott, eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3. URL: `https://doi.org/10.1007/978-3-540-71999-1`.

[15] Department of Computer Science and Electrical Engineering at UMBC. *Formal Language Definitions*. URL: `https://www.csee.umbc.edu/portal/help/theory/lang_def.shtml`.

[16] J. Cui, L. S. Liew, G. Sabaliauskaite, and F. Zhou. "A review on safety failures, security attacks, and available countermeasures for autonomous vehicles". In: *Ad Hoc Networks* 90 (2019). URL: `https://doi.org/10.1016/j.adhoc.2018.12.006`.

[17] Center for Electronic Systems Design. *Introduction to Formal Verification and Model Checking*. URL: `https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html`.

[18] T. Igarashi, M. Nakamura, and K. Sakakibara. "Formal Verification of the Lim-Jeong-Park-Lee Autonomous Vehicle Control Protocol using the OTS/CafeOBJ Method". In: July 2022, pp. 574–579. URL: `https://doi.org/10.18293/SEKE2022-028`.

[19] SRI International. *Maude software language*. URL: `https://www.sri.com/hoi/maude-software-language/`.

[20] A. Lamsweerde. "Formal specification: a roadmap". In: *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*. Ed. by A. Finkelstein. ACM, 2000, pp. 147–159. URL: `https://doi.org/10.1145/336512.336546`.

[21] J. Lim, Y. S. Jeong, D. S. Park, and H. Lee. "An efficient distributed mutual exclusion algorithm for intersection traffic control". In: *J. Super-*

comput. 74.3 (2018), pp. 1090–1107. URL: `https://doi.org/10.1007/s11227-016-1799-3`.

[22] Encyclopedia of Mathematics. *Formal language - Encyclopedia of Mathematics.* URL: `https://encyclopediaofmath.org/index.php?title=Formal_language`.

[23] J. Meseguer. “Twenty years of rewriting logic”. In: *J. Log. Algebraic Methods Program.* 81.7-8 (2012), pp. 721–781. URL: `https://doi.org/10.1016/j.jlap.2012.06.003`.

[24] W. H. H. Myint, D. D. Bui, D. D. Tran, and K. Ogata. “Graphical Animations of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol (S)”. In: *The 27th International DMS Conference on Visualization and Visual Languages, DMSVIVA 2021, KSIR Virtual Conference Center, USA, June 29-30, 2021*. Ed. by S. Chang. KSI Research Inc., 2021, pp. 22–28. URL: `https://doi.org/10.18293/DMSVIVA21-004`.

[25] D. Newcomb. *You Won’t Need a Driver’s License by 2040.* URL: `https://edition.cnn.com/2012/09/18/tech/innovation/ieee-2040-cars/index.html`.

[26] World Health Organization. *Road traffic injuries.* URL: `https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries`.

[27] Stanford Encyclopedia of Phylosophy. *Automated Reasoning.* URL: `https://plato.stanford.edu/entries/reasoning-automated/`.

[28] A. Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. URL: `https://doi.org/10.1109/SFCS.1977.32`.

[29] Embedded Staff. *An introduction to model checking.* URL: `https://www.embedded.com/an-introduction-to-model-checking/`.

Appendix

This appendix provides a list of the results and counterexamples given by **Maude**'s model-checker.

1) Counterexample for $G\neg(inIntersection(0) \land inIntersection(1))$ with initial state `init1`:

```
Maude> red modelCheck(init1, []~ (inIntersection(0) /\ inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init1, []~ (inIntersection(0) /\ inIntersection(1)))
     .
rewrites: 192 in 1ms cpu (1ms real) (163543 rewrites/second)
result ModelCheckResult: counterexample({{fstat: nEnd | istat: false | clock: 0,false,false |
     (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil)
    ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: 1,circulating,100,100},'arrive1} {{
        fstat: nEnd | istat: false | clock: 0,true,false | (l[0]: 0) ;; (l[1]:
    nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;;
         v[1]: 1,circulating,100,100},'tick} {{fstat: nEnd | istat: false |
    clock: 1,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]:
        nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 1,circulating,100,
    100},'arrive1} {{fstat: nEnd | istat: false | clock: 1,true,false | (l[0]: 0) ;; (l[1]: 1)
         ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,
    stopping,0,0) ;; v[1]: 1,stopping,1,1},'tick} {{fstat: nEnd | istat: false | clock: 2,
        false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[
    4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; v[1]: 1,stopping,1,1},'stop1} {{fstat:
        nEnd | istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (
    l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,
        stopping,1,1},'stop1} {{fstat: nEnd | istat: false | clock: 2,false,
    false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil |
         (v[0]: 0,stopped,0,0) ;; v[1]: 1,stopped,1,1},'enter1} {{fstat: nEnd |
    istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil)
        ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,crossing,0,0) ;; v[1]: 1,
    stopped,1,1},'enter2} {{fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l
        [1]: 1) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[
    0]: 0,crossing,0,0) ;; v[1]: 1,crossing,1,1},'exit1} {{fstat: nEnd | istat: true | clock:
        2,false,true | (l[0]: nil) ;; (l[1]: 1) ;; (l[2]: nil) ;; (l[3]: nil)
    ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,crossed,0,0) ;; v[1]: 1,crossing,1,1},'exit1} {{
        fstat: nEnd | istat: false | clock: 2,false,true | (l[0]: nil) ;; (l[1]:
    nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,crossed,0,0) ;;
        v[1]: 1,crossed,1,1},'end}, {{fstat: end | istat: false | clock: 2,
    false,true | (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l
        [5]: nil | (v[0]: 0,crossed,0,0) ;; v[1]: 1,crossed,1,1},deadlock})
```

2) Result for $GF(waitIntersection(0) \to inIntersection(0))$ with initial state `init2`:

```
Maude> red modelCheck(init2, ([] <> waitIntersection(0)) -> ([] <> inIntersection(0))) .
reduce in INTERSECTION-CHECK : modelCheck(init2, []<> waitIntersection(0) -> []<>
    inIntersection(0)) .
rewrites: 1077 in 3ms cpu (3ms real) (337723 rewrites/second)
result Bool: true
Maude> red modelCheck(init2, ([] <> waitIntersection(1)) -> ([] <> inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init2, []<> waitIntersection(1) -> []<>
    inIntersection(1)) .
rewrites: 1075 in 3ms cpu (3ms real) (351882 rewrites/second)
```

```
result Bool: true
```

3) Result for $GF(waitIntersection(0) \rightarrow inIntersection(0))$ with initial state `init3`:

```
Maude> red modelCheck(init3, ([] <> waitIntersection(0)) -> ([] <> inIntersection(0))) .
reduce in INTERSECTION-CHECK : modelCheck(init3, []<> waitIntersection(0) -> []<>
    inIntersection(0)) .
rewrites: 1339 in 3ms cpu (3ms real) (384770 rewrites/second)
result Bool: true
Maude> red modelCheck(init3, ([] <> waitIntersection(1)) -> ([] <> inIntersection(1))) .
reduce in INTERSECTION-CHECK : modelCheck(init3, []<> waitIntersection(1) -> []<>
    inIntersection(1)) .
rewrites: 1339 in 2ms cpu (3ms real) (485672 rewrites/second)
result Bool: true
```

4) Counterexample for $GFexecutionEnd$ with initial state `init5` that demonstrates an error in the model:

```
Maude> red modelCheck(init5, [] <> executionEnd) .
reduce in INTERSECTION-CHECK : modelCheck(init5, []<> executionEnd) .
rewrites: 14317 in 8ms cpu (8ms real) (1625269 rewrites/second)
result ModelCheckResult: counterexample({{fstat: nEnd | istat: false | clock: 0,false,false |
    (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil)
    ;; l[5]: nil | (v[0]: 0,circulating,100,100) ;; (v[1]: 2,circulating,100,100) ;; (v[2]: 3,
        circulating,100,100) ;; v[3]: 4,circulating,100,100},'arrive1} {{fstat:
    nEnd | istat: false | clock: 0,true,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l
        [3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; (v[1]:
    2,circulating,100,100) ;; (v[2]: 3,circulating,100,100) ;; v[3]: 4,circulating,100,100},'
        tick} {{fstat: nEnd | istat: false | clock: 1,false,false | (l[0]: 0) ;;
    (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping
        ,0,0) ;; (v[1]: 2,circulating,100,100) ;; (v[2]: 3,circulating,100,100)
    ;; v[3]: 4,circulating,100,100},'arrive1} {{fstat: nEnd | istat: false | clock: 1,true,
        false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]:
    nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,
        circulating,100,100) ;; v[3]: 4,circulating,100,100},'tick} {{fstat: nEnd |
    istat: false | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil)
        ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; (v[1]: 2,
    stopping,1,1) ;; (v[2]: 3,circulating,100,100) ;; v[3]: 4,circulating,100,100},'arrive1}
        {{fstat: nEnd | istat: false | clock: 2,true,false | (l[0]: 0) ;; (l[1]:
    nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,stopping,0,0) ;; (v
        [1]: 2,stopping,1,1) ;; (v[2]: 3,stopping,2,2) ;; v[3]: 4,circulating,
    100,100},'tick} {{fstat: nEnd | istat: false | clock: 3,false,false | (l[0]: 0) ;; (l[1]:
        nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: nil) ;; l[5]: nil | (v[0]: 0,
    stopping,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopping,2,2) ;; v[3]: 4,circulating
        ,100,100},'arrive1} {{fstat: nEnd | istat: false | clock: 3,true,false |
    (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]: 0,
        stopping,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopping,2,2) ;; v[
    3]: 4,stopping,3,3},'tick} {{fstat: nEnd | istat: false | clock: 4,false,false | (l[0]: 0)
        ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (
    v[0]: 0,stopping,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopping,2,2) ;; v[3]: 4,
        stopping,3,3},'stop1} {{fstat: nEnd | istat: false | clock: 4,false,false |
    (l[0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]: 0,
        stopped,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopping,2,2) ;; v[3]:
    4,stopping,3,3},'stop1} {{fstat: nEnd | istat: false | clock: 4,false,false | (l[0]: 0) ;;
         (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[
    0]: 0,stopped,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopped,2,2) ;; v[3]: 4,stopping
        ,3,3},'stop1} {{fstat: nEnd | istat: false | clock: 4,false,false | (l[
    0]: 0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]: 0,
        stopped,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopped,2,2) ;; v[3]: 4,
    stopped,3,3},'enter1} {{fstat: nEnd | istat: true | clock: 4,false,false | (l[0]: 0) ;; (l
        [1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]:
    0,crossing,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,stopped,2,2) ;; v[3]: 4,stopped
        ,3,3},'enter2} {{fstat: nEnd | istat: true | clock: 4,false,false | (l[0]:
    0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]: 0,crossing
        ,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,crossing,2,2) ;; v[3]: 4,
    stopped,3,3},'enter2} {{fstat: nEnd | istat: true | clock: 4,false,false | (l[0]: 0) ;; (l
        [1]: nil) ;; (l[2]: 1) ;; (l[3]: 2) ;; (l[4]: 3) ;; l[5]: nil | (v[0]:
    0,crossing,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,crossing,2,2) ;; v[3]: 4,crossing
        ,3,3},'exit1} {{fstat: nEnd | istat: true | clock: 4,false,false | (l[0]:
    0) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]: 3) ;; l[5]: nil | (v[0]: 0,
        crossing,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,crossed,2,2) ;; v[3]: 4,
    crossing,3,3},'exit1} {{fstat: nEnd | istat: true | clock: 4,false,false | (l[0]: 0) ;; (l
        [1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[
    0]: 0,crossing,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,crossed,2,2) ;; v[3]: 4,crossed
        ,3,3},'exit1}, {{fstat: nEnd | istat: false | clock: 4,false,true | (l[
    0]: nil) ;; (l[1]: nil) ;; (l[2]: 1) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]: nil | (v[0]:
        0,crossed,0,0) ;; (v[1]: 2,stopping,1,1) ;; (v[2]: 3,crossed,2,2) ;; v[
    3]: 4,crossed,3,3},deadlock})
```

5) Total solutions to the query `vu-narrow (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil) ;; (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0, circulating, 100, 100) ;; (v[1]: X:Nat, circulating, 100, 100) =>* (fstat: nEnd) | (istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;; (l[5]: NL5:NatList) | (v[0]: 0, stopped, T1:Nat, T1':Nat) ;; (v[1]: X:Nat, crossing, T2:Nat, T2':Nat) .`

```
Maude> vu-narrow { (fstat: nEnd) | (istat: false) | (clock: 0, false, false) | (l[0]: nil) ;;
    (l[1]: nil) ;; (l[2]: nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; (l[5]: nil) | (v[0]: 0,
    circulating, 100, 100) ;; (v[1]: X:Nat, circulating, 100, 100) } =>* { (fstat: nEnd) | (
    istat: true) | (clock: T:Nat, B:Bool, B':Bool) | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList
    ) ;; (l[2]: NL2:NatList) ;; (l[3]: NL3:NatList) ;; (l[4]: NL4:NatList) ;; (l[5]: NL5:
    NatList) | (v[0]: 0, stopped, T1:Nat, T1':Nat) ;; (v[1]: X:Nat, crossing, T2:Nat, T2':Nat)
     } .
vu-narrow in INTERSECTION-RL : {fstat: nEnd | istat: false | clock: 0,false,false | ((((l[4]:
     nil) ;; l[5]: nil) ;; l[3]: nil) ;; l[2]:
    nil) ;; l[1]: nil) ;; l[0]: nil | (v[0]: 0,circulating,100,100) ;; v[1]: X:Nat,circulating
        ,100,100} =>* {fstat: nEnd | istat: true |
    clock: T:Nat,B,B' | (l[0]: NL0:NatList) ;; (l[1]: NL1:NatList) ;; (l[2]: NL2:NatList) ;; (
        l[3]: NL3:NatList) ;; (l[4]: NL4:NatList)
    ;; l[5]: NL5:NatList | (v[0]: 0,stopped,T1:Nat,T1':Nat) ;; v[1]: X:Nat,crossing,T2:Nat,T2
        ':Nat} .

Solution 1
rewrites: 2583 in 628ms cpu (810ms real) (4107 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,crossing,1,1}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 2
rewrites: 2654 in 643ms cpu (837ms real) (4123 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
```

```
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,crossing,1,1}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 3
rewrites: 2869 in 682ms cpu (885ms real) (4205 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 3,crossing,1,1}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 4
rewrites: 2940 in 697ms cpu (904ms real) (4215 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 3,crossing,1,1}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 5
rewrites: 3011 in 712ms cpu (923ms real) (4225 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 4,crossing,1,1}
accumulated substitution:
X:Nat --> 4
variant unifier:
```

```
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 6
rewrites: 3082 in 729ms cpu (974ms real) (4221 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
     nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
     nil | (v[0]: 0,stopped,0,0) ;; v[1]: 4,crossing,1,1}
accumulated substitution:
X:Nat --> 4
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 7
rewrites: 3387 in 791ms cpu (1070ms real) (4280 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
     nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
     nil | (v[0]: 0,stopped,0,0) ;; v[1]: 1,crossing,1,1}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 8
rewrites: 3536 in 825ms cpu (1293ms real) (4282 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
     nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
     nil | (v[0]: 0,stopped,0,0) ;; v[1]: 3,crossing,1,1}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
```

```
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 9
rewrites: 3610 in 844ms cpu (1333ms real) (4274 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
    nil | (v[0]: 0,stopped,0,0) ;; v[1]: 4,crossing,1,1}
accumulated substitution:
X:Nat --> 4
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 0
T1':Nat --> 0
T2:Nat --> 1
T2':Nat --> 1

Solution 10
rewrites: 3966 in 944ms cpu (1461ms real) (4198 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 1,crossing,0,0}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 11
rewrites: 4037 in 958ms cpu (1477ms real) (4210 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 1,crossing,0,0}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
```

```
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 12
rewrites: 4108 in 974ms cpu (1496ms real) (4217 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: 1) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 1,crossing,0,0}
accumulated substitution:
X:Nat --> 1
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> 1
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 13
rewrites: 4387 in 1039ms cpu (1574ms real) (4219 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 3,crossing,0,0}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 14
rewrites: 4458 in 1054ms cpu (1593ms real) (4228 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 3,crossing,0,0}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
```

```
T2':Nat --> 0

Solution 15
rewrites: 4529 in 1069ms cpu (1615ms real) (4233 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: 1) ;; (l[4]: nil) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 3,crossing,0,0}
accumulated substitution:
X:Nat --> 3
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> 1
NL4:NatList --> nil
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 16
rewrites: 4606 in 1091ms cpu (1643ms real) (4221 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 4,crossing,0,0}
accumulated substitution:
X:Nat --> 4
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 17
rewrites: 4677 in 1106ms cpu (1667ms real) (4225 rewrites/second)
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 4,crossing,0,0}
accumulated substitution:
X:Nat --> 4
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

Solution 18
rewrites: 4748 in 1121ms cpu (1788ms real) (4233 rewrites/second)
```

```
state: {fstat: nEnd | istat: true | clock: 2,false,false | (l[0]: 0) ;; (l[1]: nil) ;; (l[2]:
    nil) ;; (l[3]: nil) ;; (l[4]: 1) ;; l[5]:
    nil | (v[0]: 0,stopped,1,1) ;; v[1]: 4,crossing,0,0}
accumulated substitution:
X:Nat --> 4
variant unifier:
T:Nat --> 2
B --> false
B' --> false
NL0:NatList --> 0
NL1:NatList --> nil
NL2:NatList --> nil
NL3:NatList --> nil
NL4:NatList --> 1
NL5:NatList --> nil
T1:Nat --> 1
T1':Nat --> 1
T2:Nat --> 0
T2':Nat --> 0

No more solutions.
rewrites: 21511 in 5180ms cpu (6532ms real) (4152 rewrites/second)
```