Formal Specification and Model Checking of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol

Moe Nandi Aung
Faculty of Science
Univ. of Info. Tech. (UIT)
Hlaing Township, PO 11052, Yangon, Myanmar
Email: moenandiaung@uit.edu.mm

Yati Phyo, Kazuhiro Ogata
School of Information Science
Japan Adv. Inst. of Sci. & Tech. (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {yatiphyo,ogata}@jaist.ac.jp

***Abstract*—We have conducted a case study in which an autonomous vehicle intersection control protocol is formally specified in Maude and model checked with Maude model checking facilities. We found that a function used in the protocol should be revised while formally specifying it and a logical clock such that times are total order should be used to avoid deadlock states during model checking experiments.**

***Keywords*-Autonomous vehicle; Intersection control; LTL; Maude; Model checking**

I. Introduction

It is predicted that vehicles would be fully autonomous (https://www.wired.com/2012/09/ieee-autonomous-2040/). To this end, multiple new technologies should be emerged and used. One of them is a protocol to control vehicles on an intersection such that collision must be avoided and vehicles can eventually cross the intersection. Lim, Jeong, Park & Lee have proposed such a protocol [1], which is called the LJPL protocol in this paper. Because such a protocol is intricate, formal methods should be utilized to formally verify that the protocol enjoys desired properties.

We have conducted a case study in which the LJPL protocol is formally specified in Maude [2], a rewriting logic-based specification/programming language, and model checked with the Maude model checking facilities (a reachability analyzer and an LTL model checker). While carefully reviewing the protocol, we found that one function used in the protocol should be revised. We also found that a logical clock such that times are total order should be used to avoid deadlock states during model checking experiments.

Related studies have been conducted, two among which are mentioned. Ölveczky and Meseguer [3] have reported on a case study in which a distributed embedded system (DES) family has been specified and model checked in Real-Time Maude, where the DES is for a pedestrian and car 4-way traffic intersection in which autonomous devices communicate by asynchronous message passing without a centralized controller. Asplund, et al. [4] have demonstrated how a Satisfiability Modulo Theory (SMT) solver can be used to prove that a distributed coordination protocol for autonomous vehicles satisfies the intersection collision avoidance property. Z3 has been used as an SMT solver. Our approach uses a high abstract notion of time to formally specify the LJPL protocol, contributing to reduction of reachable state space sizes. Thus, it is unnecessary to use any realtime-related concepts and techniques.

The rest of the paper is organized as follows: § II Preliminaries, § III Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol, § IV Formal Specification, § V Model Checking, and § VI Conclusion.

II. Preliminaries

A Kripke structure K is $\langle S, I, T, P, L\rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a total binary relation over S, P is a set of atomic propositions and $L \in S \rightarrow 2^P$ is a labeling function. $(s, s') \in T$ is called a state transition from s to s' and T may be called the state transitions. For $s \in S$, $L(s)$ is the set of atomic propositions that hold in s. The semantics of LTL is defined over infinite sequences of states generated from K. The LTL formula used in the paper is in the form $\Diamond\ p$, where p is a state formula that does not have any temporal connectives. K satisfies $\Diamond\ p$ iff for all $s_0 \in I$, for all infinite sequencers of state starting with s_0, there exists a state in the sequence such that p holds.

There are multiple possible ways to express states. We express a state as a braced associative-commutative (AC) collection of name-value pairs. AC collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a braced soup of observable components. The juxtaposition operator is used as the constructor of soups. Let oc_1, oc_2, oc_3 be observable components, and then $oc_1\ oc_2\ oc_3$ is the soup of those three observable components. A state is expressed as $\{oc_1\ oc_2\ oc_3\}$. There are multiple possible ways to specify state transitions. We specify them as rewrite rules. Concretely, we use Maude [2], a programming/specification

This work was partially supported by JSPS KAKENHI Grant Number JP26240008 & JP19H04082.
DOI reference number: 10.18293/SEKE2019-021

language based on rewriting logic. Maude makes it possible to specify complex systems flexibly and is also equipped with model checking facilities (a reachability analyzer and an LTL model checker). Maude allows us to use what are called matching equations in the conditional part of a conditional rewrite rule. A conditional rewrite rule (or just a rule) is in the form `crl [`lb`] :` l `=>` r `if` $\ldots$ `/\` p_1 `:=` p_2 `/\` $\ldots$, where lb is the label given to the rule and p_1 `:=` p_2 is a matching equation, where p_1 may have fresh variables and p_2 does not. p_1 `:=` p_2 holds if p_1 matches p_2, making a substitution in which fresh variables in p_1 are mapped to some terms (values) in p_2. Matching equations are similar to **let** expressions in functional programming languages.

The search command tries to find a state reachable from t such that the state matches p and satisfies c:

`search [1] in` M `:` t `=>*` p `such that` c `.`

where M is a specification of the S and T parts of K. t typically represents an initial state of K. The search command can be used to model check that K enjoys an invariant property if p and c represent the negation of the state formula concerned.

The search command can also be used to find a deadlock state:

`search [1] in` M `:` t `=>!` p `.`

Let $init$ be the only initial state of K and φ be an LTL formula, such as $\Diamond\ p$. Then, the Maude LTL model checker checks that K satisfies φ by reducing `modelCheck(`$init$`,`φ`)`.

III. Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol

Let us consider the intersection shown in Fig. 1, where two streets are crossed. Vehicles are supposed to run on the right-hand side of a street and each side of a street has two lanes. Each lane is named as shown in Fig. 1. When crossing the intersection, vehicles on the right lane of one side of a street, such as lane0, are supposed to go straight or turn right as shown in Fig. 1 and those on the left lane of one side of a street, such as lane1, are supposed to turn left as shown in Fig. 1. The space overlapped by the two streets is a critical section as shown in Fig. 1, where vehicles should be controlled so that they never crash into each other.

Vehicles on lane0 and those on lane4 are allowed to go through the intersection simultaneously, while vehicles on lane0 and those on lane2 are not. lane0 and lane4 are concurrent, while lane0 and lane2 are conflict. For $i = 0, 2, 4, 6$, the conflict lanes of lane$i$ are lane$j$ for $j = (i+2) \bmod 8, (i+5) \bmod 8, (i+6) \bmod 8, (i+7) \bmod 8$, and the concurrent lanes of lane$i$ are lane$j$ for $j = (i+1) \bmod 8, (i+3) \bmod 8, (i+4) \bmod 8$. For $i = 1, 3, 5, 7$, the conflict lanes of lane$i$ are lane$j$ for $j = (i+1) \bmod 8, (i+2) \bmod 8, (i+3) \bmod 8, (i+6) \bmod 8$, and the concurrent lanes of lane$i$ are lane$j$ for $j = (i+4) \bmod 8, (i+5) \bmod 8, (i+7) \bmod 8$.

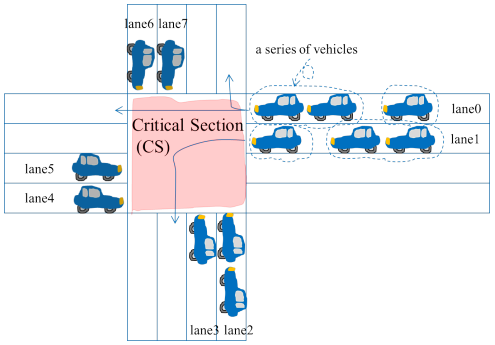

Figure 1. An intersection

Each vehicle is given its status, which is running, approaching, stopped, crossing or crossed. Note that running and approaching are not used in the paper [1] in which the LJPL protocol is proposed. When a vehicle is far enough from the intersection[1], its status is running. Each lane is given a queue. When vehicles are approaching the intersection shortly enough, their statuses become approaching, their IDs are enqueued into a queue specific to each lane, and they never change the lane and never pass the vehicles in front of them. Vehicles then estimate the time when they are supposed to get to the intersection. The paper [1] in which the the LJPL protocol is proposed says that "*To control the intersection traffic correctly, we assume that the time for vehicles is synchronized by retrieving GPS time or using a logical clock*." The present paper assumes that GPS is retrieved to synchronize vehicles, namely that there is a global clock shared by all vehicles. When a vehicle is enqueued into a queue and there is no other vehicle whose status is stopped in front on it (note that there are two such cases: (1) the vehicle is the top of the queue, namely that there is no other vehicle in front of the vehicle on the lane and (2) there is another vehicle that is followed by the vehicle and whose status is crossing on the lane), the vehicle becomes lead and its status becomes stopped. Otherwise, the vehicle becomes non-lead and its status becomes stopped. Any lead vehicle checks if there is no vehicle on any conflict lanes crossing the intersection and the time given to the vehicle is earlier than those given to the lead vehicles on the conflict lanes. If so, the lead vehicle is allowed to go through the intersection and its status becomes crossing. At the same time all non-lead vehicles whose statuses are stopped and that follow the lead vehicle are also allowed to go through the intersection and their statuses become crossing. The LJPL protocol treats a series of vehicles as a

[1]When the LJPL protocol is implemented, we need to define "being far enough from," say, 100m or longer. In this paper, however, we leave it abstract in the specification.

train so that the series of vehicles are allowed to go through the intersection simultaneously. For example, let us consider the vehicles on lane0 shown in Fig. 1 and let us suppose that the first one is lead, the second one is non-lead, the third one is lead and there is no more. When the first one is allowed to go thorough the intersection, so is the second one and then the series of the first and second ones are treated as a train.

A lead vehicle whose status is stopped on a lane exchanges some pieces of information with the lead vehicles on the four conflict lanes. One piece of information exchanged is the time (called the arrival time in this paper) when each other vehicle will arrive at the intersection. If the lead vehicle arrival time is earlier than all the four lead vehicles' arrival times on the four conflict lanes, then the status of the lead vehicle becomes crossing. Moreover, the statuses of all follower vehicles whose statuses are stopped also become crossing. When a vehicle has crossed the intersection, the status of the vehicle becomes crossed and its ID is dequeued from the queue. Such exchanges of pieces of information among lead vehicles are conducted by Algorithm 1 and Algorithm 2 [1]:

Algorithm 1. Basic IVC protocol (active thread)
1: **begin** at each round
2: $vehicle_{\text{target}} \leftarrow \text{selectVehicle}()$;
3: $\text{send}(information_{\text{local}}, vehicle_{\text{target}})$;
4: $information_{\text{target}} \leftarrow \text{receive}(vehicle_{\text{target}})$;
5: $\text{updateInformation}(information_{local}, information_{\text{target}})$;
6: **end**

Algorithm 2. Basic IVC protocol (passive thread)
1: **repeat**
2: $vehicle_{target} \leftarrow \text{waitForVehicle}()$;
3: $information_{\text{target}} \leftarrow \text{receive}(vehicle_{\text{target}})$;
4: $\text{updateInformation}(information_{\text{local}}, information_{\text{target}})$;
5: $\text{send}(information_{\text{local}}, vehicle_{\text{target}})$;
6: **until**forever;

where IVS stands for inter-vehicle-communication [5]. $information_x$, where $x = \text{target}, \text{local}$, consists of the following pieces of information:

- $lane$ – Lane number from 0 to 7
- $arrivalTime$ – Arrival time for its own vehicle
- $arrivalTime_{lead}$ – Arrival time for lead vehicle
- $lead$ – True or false
- $conflictLane$ – List of conflict lanes
- $concurrentLane$ – List of concurrent lanes
- $concurrentLanePassing$ – List of concurrent lanes for passing vehicles
- $status$ – stopped, passing, or passed

Note that running and approaching are also used as status values in this paper.

The LJPL protocol itself is described as Algorithm 3 [1]:

Algorithm 3. Mutual exclusion algorithm via IVC
1. **begin** initialization
2. $infoVehicles_i[j] \leftarrow \text{null}, \forall i \in \{1, \ldots, \max_{\text{lane}}\}, \forall j \in \{1, \ldots, \max_{\text{vehicle}}\}$;
3. **end**
4. **begin** when entering the intersection
6. $lane \leftarrow \text{getLaneNum}()$;
7. $arrivalTime \leftarrow \text{getCurrentTime}()$;
8. **if** no vehicle on the lane, where $status == \text{stopped}$ **then**
9. $lead \leftarrow \text{true}$;
10. $arrivalTimelead \leftarrow arrivalTime$;
11. **else**
12. $lead \leftarrow \text{false}$;
13. **endif**
14. $status \leftarrow \text{stopped}$;
15. **end**
16. **begin** at each cycle
17. update $infoVehicles_i[j]$ according to Algorithm 1 and Algorithm 2;
18. check $infoVehicles_{conflictLane}$ for passing the intersection;
19. **if** passingCondition() **then**
20. $status \leftarrow \text{passing}$;
21. move and cross the intersection;
22. **endif**
23. **end**
24. **begin** when exiting the intersection
25. $status \leftarrow \text{passed}$;
26. **end**
27. **function** passingCondition()
28. **if** $\forall arrivalTime_{lead} \in infoVehicles_{conflictLane} > arrivalTime_{lead}$ **and** $\forall status \in infoVehicles_{conflictLane} == \text{stopped}$ **then**
29. **return** true;
30. **else if** $\exists status \in infoVehicles_{concurrentLane} == \text{passing}$ **and** $\exists! lane_i \in \{lane_n, \forall n \in \{0, \ldots, \max_{\text{lane}}\} \mid status == \text{passing}\}$ **and** $\forall arrivalTime_{lead} \in infoVehicles_{concurrentLanePassing} > arrivalTime_{lead}$ **then**
31. **return** true;
32. **else**
33. **return** false;
34. **end function**

From a vehicle v point of view such that v is located on a lane l, function passingCondition() returns true even if there exists a lead vehicle v' on a conflict lane of l such that the status is passing (namely that the first condition is

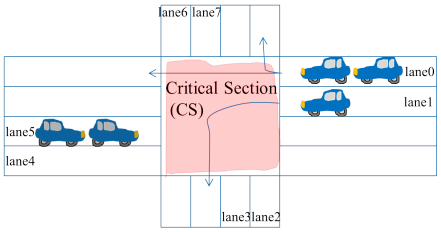

Figure 2. An initial state

false) but if the second condition is true. If so, v is permitted to cross the intersection and may crash into v'. Therefore, passingCondition() is revised as follows: it returns true if the first condition is true and returns false otherwise.

IV. Formal Specification

Let K_{LJPL} be the Kripke structure formalizing the LJPL protocol.

The four kinds of observable components are used:

- `(v[`vid`]:`lid`,`$vstat$`,`t`,`lt`)` – vid is a vehicle ID (a natural number), lid is a lane ID (a natural number) on which the vehicle vid is located, $vstat$ is the status of the vehicle vid, t is the time when the vehicle vid will reach the intersection and lt is the time when the lead vehicle will reach the intersection if any.
- `(lane[`lid`]:`q`)` – lid is a lane ID (a natural number) and q is a queue of vehicle IDs (natural numbers).
- `(clock:`t`,`b`)` – t is a natural number that represents an abstract notion of the current time and b is a Boolean value. If time is allowed to increase without any constraints, the reachable state space can quickly explode. Therefore, the following constraint will be put: whenever b is true, t can increment and b becomes false if so. When a vehicle obtains the current time t (which does not change t, though), b becomes true.
- `(gstat:`$gstat$`)` – $gstat$ is either `fin` or `nFin`. If it is `fin`, then all vehicles concerned have crossed the intersection.

Let us consider the initial state as shown in Fig. 2 such that two vehicles are running on lane0, one vehicle is running on lane1, two vehicles are running on lane5 and there is no vehicle on the other lanes. The initial state (referred as `init`) is expressed as follows:

```
{(gstat: nFin) (clock: 0,false)
 (lane[0]: oo) (lane[1]: oo) (lane[2]: oo)
 (lane[3]: oo) (lane[4]: oo) (lane[5]: oo)
 (lane[6]: oo) (lane[7]: oo)
 (v[oo]: 0,stopped,oo,oo) (v[oo]: 1,stopped,oo,oo)
 (v[oo]: 2,stopped,oo,oo) (v[oo]: 3,stopped,oo,oo)
 (v[oo]: 4,stopped,oo,oo) (v[oo]: 5,stopped,oo,oo)
 (v[oo]: 6,stopped,oo,oo) (v[oo]: 7,stopped,oo,oo)
 (v[0]: 0,running,oo,oo) (v[1]: 0,running,oo,oo)
 (v[2]: 1,running,oo,oo) (v[3]: 5,running,oo,oo)
 (v[4]: 5,running,oo,oo) }
```

Each queue for each lane, such as the one represented by `(lane[0]: oo)`, only consists of `oo` that represents ∞, which means that there is no vehicle on the lane close enough to the intersection. There are eight `v[oo]` observable components that are used to represent dummy vehicles. The `v[0]`, `v[1]`, `v[2]`, `v[3]`, and `v[4]` observable components represent the five vehicles, the first two of which are on lane0, the third of which is on lane1 and the last two of which are on lane5. The global clock represented by `(clock: 0,false)` is initially 0. Because the second value of the `clock` observable component is `false`, the abstract notion of the current time cannot increment. The value of the `gstat` observable component is initially `nFin`.

T_{LJPL} is specified by 12 rewrite rules. The following rule is used to make T_{LJPL} total:

```
rl [stutter] : {(gstat: fin) OCs}
=> {(gstat: fin) OCs} .
```

where `OCs` is a Maude variable of observable component soups.

```
crl [fin] : {(gstat: nFin) OCs}
=> {(gstat: fin) OCs} if fin?(OCs) .
```

where `fin?(OCs)` returns true if and only if each vehicle in `OCs` has crossed the intersection.

```
rl [tick] :
{(gstat: nFin) (clock: T,true) OCs}
=>
{(gstat: nFin) (clock: (T + 1),false) OCs} .
```

where `T` is a Maude variable of natural numbers. If the second value of the `clock` observable component is `true`, the abstract notion of the current time `T` increments and the second value becomes `false`.

Two rules are used to specify a set of transitions that change a vehicle status from running to approaching. One rule deals with the case in which there is no vehicle close enough to the intersection on the lane where the vehicle is running, and the other deals with the case in which there exists at least one vehicle close enough to the intersection on the lane. The two rules are as follows:

```
rl [approach1] :
{(gstat: nFin) (clock: T,B) (lane[LI]: oo)
 (v[VI]: LI,running,oo,oo) OCs}
=>
{(gstat: nFin) (clock: T,true)
 (lane[LI]: VI) (v[VI]: LI,approaching,T,oo)
 OCs} .

rl [approach2] : {(gstat: nFin) (clock: T,B)
 (lane[LI]: (VI' ; VS))
 (v[VI]: LI,running,oo,oo) OCs}
=> {(gstat: nFin) (clock: T,true)
 (lane[LI]: (VI' ; VS ; VI))
 (v[VI]: LI,approaching,T,oo) OCs} .
```

where `B` is Maude variable of Boolean values, `LI`, `VI` & `VI'` are Maude variables of natural numbers, `VS` is a

Maude variable of queues of natural numbers & ∞, and _;_ is the constructor of queues, where an underscore _ is a place holder where an argument is put. _;_ is associative, meaning that $(q_1 \ ; \ q_2) \ ; \ q_3 = q_1 \ ; \ (q_2 \ ; \ q_3)$, and a single element (a natural number or ∞) is treated as the singleton queue that only consists of the element.

Three rules are used to specify a set of transitions that change a vehicle status from approaching to stopped. The first rule `check1` deals with the case in which the vehicle is the top of the queue concerned, where the vehicle will be lead on the lane concerned, the second rule `check2` deals with the case in which there exists a vehicle in front of the vehicle on the lane concerned such that the preceding vehicle status is stopped, where the vehicle will be non-lead on the lane, and the third rule `check3` deals with the case in which there exists a vehicle in front of the vehicle on the lane concerned such that the preceding vehicle status is crossing, where the vehicle will be lead on the lane. The three rules are as follows:

```
rl [check1] :
{(gstat: nFin) (lane[LI]: (VI ; VS))
 (v[VI]: LI,approaching,T,oo) OCs}
=> {(gstat: nFin) (lane[LI]: (VI ; VS))
 (v[VI]: LI,stopped,T,T) OCs} .

rl [check2] : {(gstat: nFin)
 (lane[LI]: (VS' ; VI' ; VI ; VS))
 (v[VI']: LI,stopped,T,T')
 (v[VI]: LI,approaching,T'',oo) OCs}
=> {(gstat: nFin)
 (lane[LI]: (VS' ; VI' ; VI ; VS))
 (v[VI']: LI,stopped,T,T')
 (v[VI]: LI,stopped,T'',T') OCs} .

rl [check3] : {(gstat: nFin)
 (lane[LI]: (VS' ; VI' ; VI ; VS))
 (v[VI']: LI,crossing,T,T')
 (v[VI]: LI,approaching,T'',oo) OCs}
=> {(gstat: nFin)
 (lane[LI]: (VS' ; VI' ; VI ; VS))
 (v[VI']: LI,crossing,T,T')
 (v[VI]: LI,stopped,T'',T'') OCs} .
```

where `T'` & `T''` are Maude variables of natural numbers and `VS'` is a Maude variable of queues. The reason why `v[`vid`]` observable components, where vid is a vehicle ID, do not have any values that correspond to $lead$ in $information_x$ is that it is possible to use queues, etc. to manage which vehicles are lead or not.

Two rules are used to specify a set of transitions that change a lead vehicle status from stopped to crossing. One rule deals with the case in which the ID of the lane on which the lead vehicle is located is even and the other deals with the case in which it is odd. The first rule `enter1` is as follows:

```
crl [enter1] :
{(gstat: nFin) (lane[LI]: (VI ; VS))
 (v[VI]: LI,stopped,T,T) OCs}
=> {(gstat: nFin) (lane[LI]: (VI ; VS))
 (v[VI]: LI,crossing,T,T) OCs'}
if isEven(LI) /\
   LI1 := (LI + 2) rem 8 /\
   (lane[LI1]: (VI1 ; VS1))
   (v[VI1]: LI1,VSt1,T11,T12) OCs1 := OCs /\
   VSt1 = stopped /\ T < T12 /\
   LI2 := (LI + 5) rem 8 /\
   (lane[LI2]: (VI2 ; VS2))
   (v[VI2]: LI2,VSt2,T21,T22) OCs2 := OCs /\
   VSt2 = stopped /\ T < T22 /\
   LI3 := (LI + 6) rem 8 /\
   (lane[LI3]: (VI3 ; VS3))
   (v[VI3]: LI3,VSt3,T31,T32) OCs3 := OCs /\
   VSt3 = stopped /\ T < T32 /\
   LI4 := (LI + 7) rem 8 /\
   (lane[LI4]: (VI4 ; VS4))
   (v[VI4]: LI4,VSt4,T41,T42) OCs4 := OCs /\
   VSt4 = stopped /\ T < T42 /\
   OCs' := letCross(VS,OCs) .
```

where `LI`i for i =1,...,4 is a Maude variable of natural numbers, `VI`i & `T`j for i =1,...,4 & j =11,12,...,41,42 are Maude variables of natural numbers & ∞, `VS`i for i =1,...,4 is a Maude variable of queues, `VSt`i for i =1,...,4 is a Maude variable of vehicle statuses, and `OCs`i & `OCs'` for i =1,...,4 are Maude variables of observable component soups. `isEven(LI)` holds if `LI` is even. For the front-most vehicle (which may be a dummy one whose ID is `oo`) of each of the four conflict lanes of lane `LI`, the rules checks if it is not crossing the intersection and its arrival time is greater than the arrival time `T` of the vehicle `VI` concerned. If the conditions are fulfilled, the status of the vehicle `VI` concerned becomes crossing from stopped and the statuses of all vehicles that follow `VI` and whose statuses are stopped also become crossing, which is done by `letCross(VS,OCs)`. The second rule `enter2` can be described likewise.

Two rules are used to specify a set of transitions that change a vehicle status to crossed from crossing. The first rule deals with the case in which the vehicle concerned is one and only one vehicle in the queue that corresponds to the lane concerned, and the second rule deals with the case in which the queue that corresponds to the lane concerned contain two or more vehicles. The two rules are as follows:

```
rl [leave1] :
{(gstat: nFin) (lane[LI]: VI)
 (v[VI]: LI,crossing,T,T') OCs}
=> {(gstat: nFin) (lane[LI]: oo)
 (v[VI]: LI,crossed,T,T') OCs} .

rl [leave2] :
{(gstat: nFin) (lane[LI]: (VI ; VI' ; VS))
 (v[VI]: LI,crossing,T,T') OCs}
=> {(gstat: nFin) (lane[LI]: (VI' ; VS))
 (v[VI]: LI,crossed,T,T') OCs} .
```

When a vehicle status changes to closed from crossing, it is deleted from the queue that corresponds to the lane concerned.

V. Model Checking

The paper [1] in which the LJPL protocol is proposed takes into account three desired properties the protocol should enjoy:

- *Safety (version 2) No vehicles in conflict lanes are in CS at the same time.*
- *Deadlock-freedom If a vehicle is trying to pass the intersection (CS), then some vehicle, not necessarily the same one, finally pass the intersection.*
- *Starvation-freedom If a vehicle is trying to pass the intersection (CS), then the vehicle must finally pass the intersection in finite time.*

The *Safety (version 2)* property can be checked by the following search command:

```
search [1] in IMUTEX : init =>*
{(v[VI]: LI,crossing,T,T')
 (v[VI']: LI',crossing,T2,T2') OCs}
such that areConflict(LI,LI') .
```

where `IMUTEX` is the specification of the protocol and `areConflict(LI,LI')` holds if and only if the lanes `LI` and `LI'` are conflict. The search commands tries to find a state from the initial state `init` such that two vehicles are crossing the intersection on two conflict lanes `LI` and `LI'`. It does not find any such states, meaning that the protocol enjoys the *Safety (version 2)* property for the case as shown in Fig. 2.

The *Deadlock-freedom* property can be checked by the following search command:

```
search [1] in IMUTEX : init =>! {OCs} .
```

The search commands tries to find a state from the initial state `init` such that no transition can be conducted. It finds such a state that contains the following observable components:

```
(lane[0]: 0 ; 1) (lane[5]: 3 ; 4)
(v[0]: 0,stopped,0,0) (v[1]: 0,stopped,0,0)
(v[3]: 5,stopped,0,0) (v[4]: 5,stopped,0,0)
```

Vehicle `0` is the lead one on lane0, while vehicle `3` is the lead one on lane5. The both lead vehicles' arrival times are 0. Therefore, either one is not permitted to cross the intersection.

The counterexample of the *Deadlock-freedom* property implies that it does not suffice to use a global clock, which may create a symmetry. To break the symmetry, lane IDs could be used. When there are multiple lead vehicles on conflict lanes such that their arrival times are exactly the same, higher priorities are given to those on lanes whose IDs are less. That is, a logical clock such that times are total order is used. In the specification, the two rules `enter1` and `enter2` should be revised such that `T < T1`i for $i = 1, \ldots, 4$ used in the conditions is replaced with `(T < T1`i `or (T == T1`i `and LI < LI`i`))`. The protocol in which a logical clock such that times are total order is used enjoys both the *Safety (version 2)* and *Deadlock-freedom* properties for for the case as shown in Fig. 2. Let the protocol refer to the one in which a logical clock such that times are total order is used.

To model check that the protocol enjoys the *Starvation-freedom* freedom, it is necessary to define P_{LJPL} and L_{LJPL}. P_{LJPL} consists of one atomic proposition `fin`. L_{LJPL} is defined as follows:

```
eq {(gstat: fin) OCs} |= fin = true .
eq {OCs} |= PROP = false [owise] .
```

where `PROP` is a Maude variable of atomic propositions. The two equations say that for all states $s \in S_{\mathrm{LJPL}}$ $L_{\mathrm{LJPL}}(s) = \{\texttt{fin}\}$ if and only if s contains `(gstat: fin)`. The *Starvation-freedom* property is then defined as follows:

```
eq halt = <> fin .
```

where `<>` is the LTL eventually connective $\Diamond$. We model check if the protocol enjoys the *Starvation-freedom* property by reducing `modelCheck(init,halt)`. No counterexample is found. Thus, the protocol enjoys the property for the case as shown in Fig. 2.

VI. Conclusion

We have reported on a case study in which the LJPL protocol is formally specified in Maude and model checked with Maude model checking facilities. We found that a function used in the protocol should be revised while formally specifying it and a logical clock such that times are total order should be used to avoid deadlock states during model checking experiments.

We have encountered the state explosion problem when we tried to model check that the protocol enjoys some property for the case as shown in Fig. 1. One piece of our future work is to remedy the situation, say, by using the divide & conquer approach to (leads-to) model checking [6].

References

[1] J. Lim, Y. Jeong, D. Park, and H. Lee, "An efficient distributed mutual exclusion algorithm for intersection traffic control," *The Journal of Supercomputing*, vol. 74, pp. 1090–1107, 2018.

[2] Clavel, M., et al., *All About Maude*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[3] P. C. Ölveczky and J. Meseguer, "Specification and verification of distributed embedded systems: A traffic intersection product family," in *RTRTS 2010*, 2010, pp. 137–157.

[4] M. Asplund, A. Manzoor, M. Bouroche, S. Clarke, and V. Cahill, "A formal approach to autonomous vehicle coordination," in *FM 2012*, 2012, pp. 52–67.

[5] M. L. Sichitiu and M. Kihl, "Inter-vehicle communication systems: A survey," *IEEE Commun. Surveys Tuts.*, vol. 10, no. 1-4, pp. 88–105, 2008.

[6] Y. Phyo and K. Ogata, "Formal specification and model checking of the Walter-Welch-Vaidya mutual exclusion protocol for ad hoc mobile networks," in *APSEC 2018*, 2018.