Journal of Visual Language and Computing

journal homepage: www.ksiresearch.org/jvlc

Graphical Animations of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol[⋆,⋆⋆]

Dang Duy Bui[a], Win Hlaing Hlaing Myint[a], Duong Dinh Tran[a] and Kazuhiro Ogata[a,*]

[a]School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

ARTICLE INFO

Article History:
Submitted 3.1.2021
Revised 6.1.2021
Second Revision 8.1.2021
Accepted 10.21.2021

Keywords:
graphical animation
autonomous vehicle intersection protocol
state machine
state picture design

ABSTRACT

SMGA is a tool that mainly supports humans in visually perceiving the characteristics/properties of systems/protocols. Those characteristics could be used as lemmas to formally verify that the systems/protocols enjoy desired properties. The core task of the tool is to design a state picture that helps humans comprehend the systems/protocols better and then conjecture the characteristics. To demonstrate that SMGA can be applied to a wider class of systems/applications, we have graphically animated the Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol with SMGA. The state machine formalizing the protocol uses composite data. We have revised SMGA so as to handle composite data. We design a flexible state picture for the protocol so that it is possible to deal with different initial states when the number of vehicles is less than or equal to a given number. In the conference version of the present paper, the visual representations of vehicles (or vehicle statuses) on each lane did not preserve the actual order of the vehicles on the lane. We have also revised SMGA so that it is possible to make a state picture design that preserves such an order. In the conference version, any information on conflict and concurrent lanes for each lane was not displayed. We have revised the state picture design so that such information can be visualized. Some characteristics are guessed by observing graphical animations based on the state picture design, and the characteristics are confirmed with model checking. The paper also summarizes several lessons learned as tips on how to design a state picture with composite data.

1. Introduction

SMGA [15] has been developed to visualize graphical animations of systems/protocols. The main purpose of SMGA is to help human users be able to perceive non-trivial characteristics of systems/protocols by observing its graphical animations because humans are good at visual perception [10]. Those characteristics could be used as lemmas to formally verify that the systems/protocols enjoy some desired properties. It implies the usefulness of the tool since lemma conjecture is a challenging problem in formal verification. Several case studies of some protocols have been conducted with SMGA, such as Alternating Bit Protocol (ABP) [15], a communication protocol, Qlock and MCS protocols [3, 16, 5, 6] shared-memory mutual exclusion protocols, and Suzuki-Kasami protocol [4], a distributed mutual exclusion protocol, to demonstrate its power.

Nowadays, autonomous vehicles or self-driving cars are a trend of the era. They have many potentials that make humans more convenient. The Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol (the LJPL protocol) [12] is one possible way to handle traffic control management for intersections through which vehicles/cars pass. It also has been formally specified in Maude by Moe et al. [2]. Thus, it is ready to graphically animate the LJPL protocol with SMGA so as to demonstrate that SMGA can be applied to protocols of autonomous vehicles/self-driving cars. Our motivations of the work described in the present paper are two-fold: (1) we would like to show that SMGA

[⋆]This work was partially supported by FY2020 grant-in-aid for new technology research activities at universities (SHIBUYA SCIENCE CULTURE AND SPORTS FOUNDATION).

[⋆⋆]The present paper is an extended and revised version of the paper [14] presented at DMSVIVA 2021.

[*]Corresponding author

bddang@jaist.ac.jp (D.D. Bui); winhlainghlaingmyint@jaist.ac.jp (W.H.H. Myint); duongtd@jaist.ac.jp (D.D. Tran); ogata@jaist.ac.jp (K. Ogata)

ORCID(s): 0000-0002-2700-1762 (D.D. Bui); 0000-0001-7092-2084 (D.D. Tran); 0000-0002-4441-3259 (K. Ogata)

DOI reference number: 10.18293/JVLC2022-N1-004

can be applied to a wide class of protocols/systems, and (2) we would like to be prepared by conjecturing non-trivial characteristics for future formal proofs that the LJPL protocol enjoys some desired properties. The LJPL protocol was taken for motivation (1) as written. Moe et al. [2] conducted some model checking experiments for the LJPL protocol but did not do any formal proofs for the protocol. Standard model checking cannot guarantee that protocols/systems surely enjoy desired properties in general, although it is good at finding counterexamples. It would be necessary to use theorem proving so as to guarantee that protocols/systems surely enjoy desired properties. Formal proofs that the LJPL protocol enjoys desired properties is one piece of our future work for which we must use several lemmas. Motivation (2) is for this purpose.

We need to carefully make a state picture design in order to produce good graphical animations because it is considered a core task of the tool [5]. To make a state picture design, we first specify the protocol in Maude. The specification, however, contains some observable components whose values are composite (over one component value inside). It is non-trivial to deal with composite data with SMGA. One possible way to visualize such a composite value is to add extra observable components each of which stores a copy of a (component) value that composes the composite value, where observable components are what constitute states in our way to specify state machines and extra observable components play roles as auxiliary/ghost variables used in formal verification. This option may make state expressions unnecessarily complex. We did not take this option but have revised the tool so that users can design a state picture to be able to display (each component of) a composite value explicitly. Even if the number of vehicles is fixed, the LJPL protocol has multiple initial states. Given a natural number n, we make a flexible state picture design so that any initial states in which the number of vehicles is up to n can be handled. Some characteristics are then guessed by observing graphical animations generated from the design. The characteristics are also confirmed with Maude [7]. Some lessons learned are summarized as tips on how to design a good state picture with observable components whose values are composite.

In the conference version [14] of the present paper, the visual representations of vehicles (or vehicle statuses) on each lane did not preserve the actual order of the vehicles on the lane. We have also revised SMGA so that it is possible to make a state picture design that preserves such an order. In the conference version [14], any information on conflict and concurrent lanes for each lane was not displayed. We have revised the state picture design so that such information can be visualized.

The rest of the paper is organized as follows. Sect. 2 mentions some preliminaries such as state machines, Maude, and SMGA. Sect. 3 introduces the LJPL protocol. Sect. 4 describes formal specification of the LJPL protocol in Maude. In Sect 5, we describe how to graphically animate the LJPL protocol. We discuss some ideas of how to handle observable components whose values are composite. Some characteristics of the LJPL protocol are then guessed by observing its graphical animations and confirmed by model checking in Sect. 6. Sect. 7 summarizes lessons learned as tips on how to design a state picture with composite data. Sect. 8 mentions some related work. Finally, we conclude the present paper in Sect. 9.

All of the state pictures and state sequences for SMGA presented in this paper are available at https://bddang.bitbucket.io/.

2. Preliminaries

A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states, a set $I \subseteq S$ of initial states, and a binary relation $T \subseteq S \times S$ over states. $(s, s') \in T$ is called a state transition and may be written as $s \rightarrow_M s'$. The set $R \subseteq S$ of reachable states with respect to M is inductively defined as follows: (1) for each $s \in I$, $s \in R$ and (2) for each $(s, s') \in T$, if $s \in R$, then $s' \in R$. A state predicate p is an invariant property with respect to M if and only if $p(s)$ holds for all $s \in R$. A finite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$ of states is called a finite computation of M if $s_0 \in I$ and $(s_i, s_{i+1}) \in T$ for each $i = 0, \ldots, n-1$.

In this paper, to express a state of S, we use a braced associative-commutative collection of name-value pairs. Associative-commutative 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 $oc1, oc2, oc3$ be observable components, and then $oc1\ oc2\ oc3$ is the soup of those three observable components. A state is expressed as $\{oc1\ oc2\ oc3\}$. There are multiple possible ways to specify state transitions. In this paper, we use Maude [7], a programming/specification language based on rewriting logic, to specify them as rewrite rules. 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). A rewrite rule starts with the keyword `rl`, followed by a label enclosed with square brackets and a colon, two patterns (terms that may contain variables) connected with =>, and ends with a full stop. A conditional one starts with the keyword `crl` and has a condition following the keyword `if` before a full stop. The following is a form of a conditional rewrite rule:

`crl` $[lb] : l$ => r `if` … /\ c_i /\ …

where lb is a label and c_i is part of the condition, which may be an equation $lc_i = rc_i$ or a matching equation $lc_i := rc_i$. The negation of $lc_i = rc_i$ could be written as $(lc_i =/= rc_i) =$ `true`, where `= true` could be omitted. For a given subject term t, if there exist a sub-term t' of t and a substitution σ such that $t' = \sigma(l)$ and the condition … /\ c_i /\ … holds under σ, t' in t is replaced with $\sigma'(r)$, where σ' is a substitution obtained by σ and substitutions calculated by matching equations. For $lc_i := rc_i$, lc_i may have new variables that do not appear in l and the other matching equations so far, while rc_i does not have such new variables. $lc_i := rc_i$ holds if and only if there exists a substitution σ_i such that $\sigma_i(\sigma''(lc_i)) = \sigma''(rc_i)$, where σ'' is a substitution obtained by σ and substitutions calculated by the matching equations so far.

Maude provides the search command that allows finding a state reachable from t such that the state matches p and satisfies c:

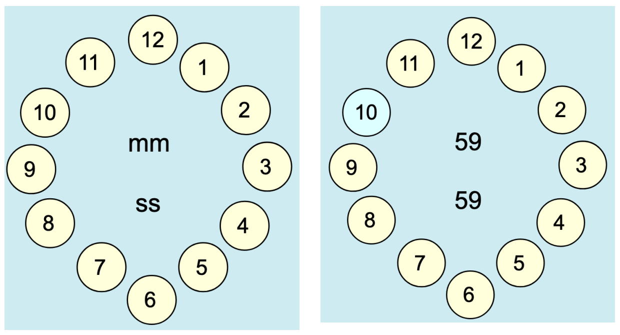
search [n,m] in MOD : t =>* p such that c .

where `MOD` is the name of the module specifying the state machine, and `n` and `m` are optional arguments stating a bound on the number of desired solutions and the maximum depth of the search, respectively. `n` typically is 1 and t typically represents an initial state of the state machine.

State Machine Graphical Animation (SMGA) is a tool developed by Nguyen and Ogata [15]. SMGA does not automatically produce visual state picture design but it allows human users to design a good state picture. An input requires a state picture designed by humans and a state sequence generated by Maude. An output is graphical animations based on series of pictures in which each state is displayed based on the state picture design. There are two ways to visualize the observable component at each state that temporarily called (1) text display and (2) mark display. For example, one state that simulates a clock is specified as follows:

```
{(hh: 10) (mm: 59) (ss: 59)}
```

where `hh`, `mm`, and `ss` are observable components receiving `10`, `59`, and `59` as their values, respectively. The following figure displays a state picture design (on the left-hand side), and a state picture (on the right-hand side) of the example in which `hh` is presented as (2) (mark display), `mm` and `ss` are presented as (1) (text display).



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

Unlike the traditional traffic light mechanism, which has a drawback in choosing the optimal time and cycles of light signals with the different numbers of vehicles in different lanes of various intersections, the LJPL protocol provides an efficient solution to manage the traffic of intersections. Suppose that vehicles run on the right-hand side of a street and each side of a street has two lanes as shown in Figure 1. We also suppose that when a vehicle is crossing the intersection, if it is running on the right lane of its moving direction, then

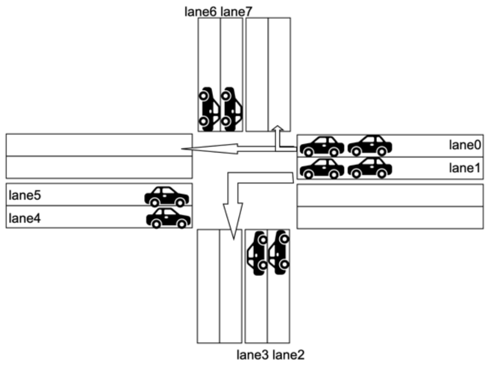


Figure 1: An example of intersection

it can only go straight or turn right. On the other hand, if the vehicle is crossing the intersection on the left lane, it can only turn left.

For each lane$i$, two relations between it and the other lanes are introduced as follows:

- *conflict lanes*: the conflict lanes of lane$i$ are lane$j$ for $j = (i+2)\ \%\ 8, (i+5)\ \%\ 8, (i+6)\ \%\ 8, (i+7)\ \%\ 8$ if $i = 0, 2, 4, 6$; and $j = (i+1)\ \%\ 8, (i+2)\ \%\ 8, (i+3)\ \%\ 8, (i+6)\ \%\ 8$ if $i = 1, 3, 5, 7$ (where % is the modulo operation). For example, lane0 and lane2 are conflict, meaning that vehicles on lane0 and those on lane2 are not allowed to go through the intersection simultaneously.

- *concurrent lanes*: the concurrent lanes of lane$i$ are lane$j$ for $j = (i+1)\ \%\ 8, (i+3)\ \%\ 8, (i+4)\ \%\ 8$ if $i = 0, 2, 4, 6$; and $j = (i+4)\ \%\ 8, (i+5)\ \%\ 8, (i+7)\ \%\ 8$ if $i = 1, 3, 5, 7$. For example, lane0 and lane4 are concurrent, meaning that vehicles on those two lanes are allowed to go through the intersection simultaneously.

Each vehicle has as its status one of the following five values: running, approaching, stopped, crossing, or crossed as its status. Let us note that Lim et al. do not use running & approaching statuses in the paper [12] and they use passing & passed statuses instead of crossing & crossed, respectively. There are eight queues of vehicles, each of them is associated with one of eight lanes. When a vehicle is far enough from the intersection, its status value is running, and when the vehicle is approaching the intersection shortly enough, its status changes to approaching, and its ID is enqueued into the queue associated with its lane. A vehicle is supposed to never change the lane and never pass the vehicles in front of it after its status changes to approaching. After a vehicles becomes approaching as its status, its status becomes stopped, which means that the vehicle stops in front of the intersection. Furthermore, the vehicle becomes lead if there is not any other vehicle whose status value is stopped in front of it; otherwise, it becomes non-lead. Note that there are two possible cases such that the vehicle becomes lead: (1) the vehicle is the top of the queue (i.e., there is no other vehicle in front of it on the lane), and (2) there is another vehicle in front of it on the lane whose status value is crossing.

Every lead vehicle checks if there is no vehicle on any conflict lane crossing the intersection and the time given to it is earlier than those given to the lead vehicles on the conflict lanes (i.e., it arrives at the intersection earlier than any others). If so, the lead vehicle is allowed to go through the intersection, and its status changes to crossing. At the same time all non-lead vehicles following the lead vehicle and statuses are stopped are also allowed to go through the intersection together with the lead and their statuses also change to crossing. When a vehicle has crossed the intersection, the status of the vehicle becomes crossed and its ID is dequeued from the queue.

Some information needs to be exchanged among lead vehicles to synchronize the protocol (e.g., comparing the arrival times). Algorithm 1 and Algorithm 2 describe how a lead vehicle whose status value is stopped on a lane exchanges information with the lead vehicles on the four conflict lanes.

Algorithm 1. Basic IVC protocol (active thread)

```
1: begin at each round
2:     $vehicle_{target}$ ← selectVehicle();
3:     send($information_{local}$,$vehicle_{target}$);
4:     $information_{target}$ ← receive($vehicle_{target}$);
5:     updateInformation($information_{local}$,
                         $information_{target}$);
6: end
```

Algorithm 2. Basic IVC protocol (passive thread)

```
1: repeat
2:     $vehicle_{target}$ ← waitForVehicle();
3:     $information_{target}$ ← receive($vehicle_{target}$);
4:     updateInformation($information_{local}$,
                         $information_{target}$);
5:     send($information_{local}$, $vehicle_{target}$);
6: until forever
```

where IVC stands for inter-vehicle-communication [12] and $information_v$ for $v = local, target$ consists of the following information:

- *lane*: Lane number from 0 to 7
- *arrivalTime*: Arrival time for its own vehicle
- *arrivalTime$_{lead}$*: Arrival time for the 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

Let us repeat again that in the present paper, we use crossing & crossed instead of passing & passed; and we add running & approaching as the *status* values.

The LJPL protocol itself is described as Algorithm 3:

Algorithm3. Mutual exclusion algorithm via IVC

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

4. Specification of LJPL Protocol in Maude

As written, in this paper, a state is expressed as a braced soup of observable components. Let b is a Boolean value, q

is a queue of vehicle IDs (i.e., a queue of natural numbers because natural numbers are used as vehicle IDs). Let vid, lid, t, lt are natural numbers, where vid and lid represent a vehicle ID and a lane ID, respectively, while t and lt represent the time. To formalize the LJPL protocol as a state machine M_{LJPL}, we use the following observable components:

- (`clock` : t, b) - it says that the current time is t. `clock` represents the global clock shared by all vehicles. Initially, the first parameter of `clock` is set to 0 and will increment. However, if time is allowed to increase without any constraints, the reachable state space will quickly explode. That is the reason why we introduce the second component b such that t only can increment when b is true. That is, whenever b is true, t can increment and b becomes false, and when a vehicle obtains the current time t (without changing t), b becomes true.

- (`v`$[vid]$: $lid, vstat, t, lt$) - it says that the vehicle vid is running on the lane lid, its current *status* is $vstat$, it arrives at the intersection at the time t, and the lead vehicle of the lane lid reaches the intersection at the time lt.

- (`lane`$[lid]$: q) - it says that the queue of vehicles running on lane lid is q.

- (`gstat` : $gstat$) - it says whether all vehiles concerned have crossed, where $gstat$ is either `fin` or `nFin`. When it is `fin`, all vehicles concerned have crossed the intersection.

Each state in S_{LJPL} is expressed as $\{obs\}$, where obs is a soup of those observable components. We suppose that five vehicles (from 0 to 4) participate in the LJPL protocol such that two vehicles are running on lane0, one vehicle is running on lane1, and two vehicles are running on lane5. The initial state of I_{LJPL} namely `init` is defined 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[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)
(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)}
```

Initially, `gstat` is set to `nFin`, the value of the global clock is 0. Since the second value of the `clock` observable component is `false`, the abstract notion of the current time cannot increment. Each queue associated with each lane only consists of `oo` (denoting ∞), saying that there is no vehicle on the lane close enough to the intersection. `v[0]` & `v[1]` represent the two vehicles running on lane0, `v[2]` represents the vehicle running on lane1, and `v[3]` & `v[4]` represent the two vehicles running on lane5. There are eight `v[oo]` observable components that are used to represent dummy vehicles.

12 rewrite rules are used to specify T_{LJPL}. Let `OCs` and `OCs'` be Maude variables of observable component soups, `T`, `T'` and `T''` be Maude variables of natural numbers, and `B` is a Maude variable of Boolean values. When all vehicles have crossed the intersection, the state does not change anything, which is specified by the following two rewrite rules:

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

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

where `fin?(OCs)` returns true iff all vehicles in `OCs` have crossed the intersection.

The rewrite rule `tick` is defined to specify the behavior of the global clock:

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

The rewrite rule says that 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 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)
(v[VI]: LI,running,oo,oo)
(lane[LI]: (VI' ; VS)) OCs}
=> {(gstat: nFin) (clock: T,true)
(lane[LI]: (VI' ; VS ; VI))
(v[VI]: LI,approaching,T,oo) OCs} .
```

where `LI`, `VI`, and `VI'` are Maude variables of natural numbers, `VS` is a Maude variable of queues of natural numbers and ∞, and ; is the constructor of queues. Note that ; is declared as an associative binary operator and each natural number or `oo` is declared as a singleton queue. Thus, `VI' ; VS ; VI` denotes the queue obtained by putting `VI` into the queue denoted as `VI' ; VS` at the end. The first rewrite rule specifies the case in which there is no vehicle close enough to the intersection on the lane where the vehicle is running, while the second one deals with the case in which there exists at least one vehicle close enough to the intersection on the lane.

Three rewrite rules are used to specify a set of transitions that change a vehicle status from approaching to stopped as follows:

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

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

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

where `VS'` is a Maude variable of queues. The first rewrite rule specifies the case in which vehicle `VI` is the top of the queue (i.e., `VI` will be lead on the lane). The second one deals with the case in which there exists another vehicle `VI'` in front of the vehicle `VI` such that `VI'` is stopped (`VI` will be non-lead on the lane). The last one specifies the case in which there exists another vehicle `VI'` in front of the vehicle `VI` such that the *status* of `VI'` is crossing (`VI` will be lead on the lane).

Two rewrite rules `enter1` and `enter2` are used to specify a set of transitions that change a lead vehicle *status* from stopped to crossing. `enter1` deals with the case in which the ID of the lane on which the lead vehicle is located is even and `enter2` deals with the case in which it is odd. `enter1` is defined as follows:

```
crl [enter1] : {(v[VI]: LI,stopped,T,T)
(gstat: nFin) (lane[LI]: (VI ; VS)) 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) .
```

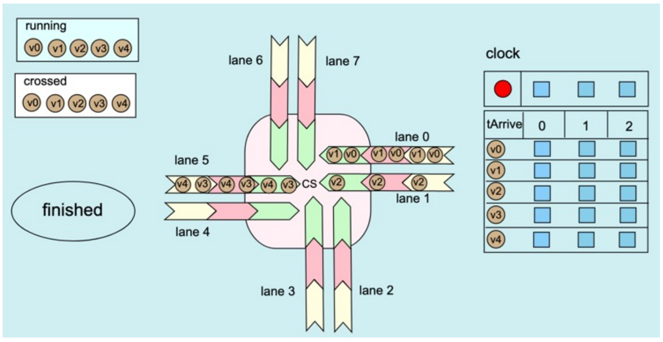
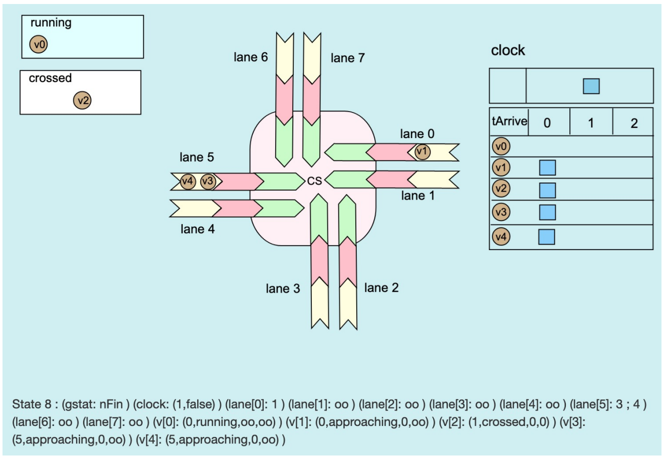


Figure 2: A state picture design for the LJPL protocol (1)



State 8 : (gstat: nFin) (clock: (1,false)) (lane[0]: 1) (lane[1]: oo) (lane[2]: oo) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 4) (lane[6]: oo) (lane[7]: oo) (v[0]: (0,running,oo,oo)) (v[1]: (0,approaching,0,oo)) (v[2]: (1,crossed,0,0)) (v[3]: (5,approaching,0,oo)) (v[4]: (5,approaching,0,oo))

Figure 3: A state picture for the LJPL protocol (1)

where `LI`i for $i = 1, \ldots, 4$ are Maude variables of natural numbers, `VI`i & `T`j for $i = 1, \ldots, 4$ & $j = 11, 12, \ldots, 41, 42$ are Maude variables of natural numbers & ∞, `VS`i for $i = 1, \ldots, 4$ are Maude variables of queues, `VSt`i for $i = 1, \ldots, 4$ are Maude variables of vehicle statuses, and `OCs`i for $i = 1, \ldots, 4$ are Maude variables of observable component soups. `isEven(LI)` holds if `LI` is even. The rewrite rule checks if all lead vehicles of the four conflict lanes (i.e., `LI1`, `LI2`, `LI3`, and `LI4`) are not crossing the intersection and the arrival time `T` of the vehicle `VI` is less than all arrival times of the lead vehicles on the conflict lanes. If the conditions are satisfied, the status of vehicle `VI` is changed to 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 rewrite rule `enter2` can be defined likewise. There are also two more rewrite rules `leave1` and `leave2` that are used to specify a set of transitions changing a vehicle *status* to crossed from crossing. All of them can be found from the webpage presented in Sect. 1.

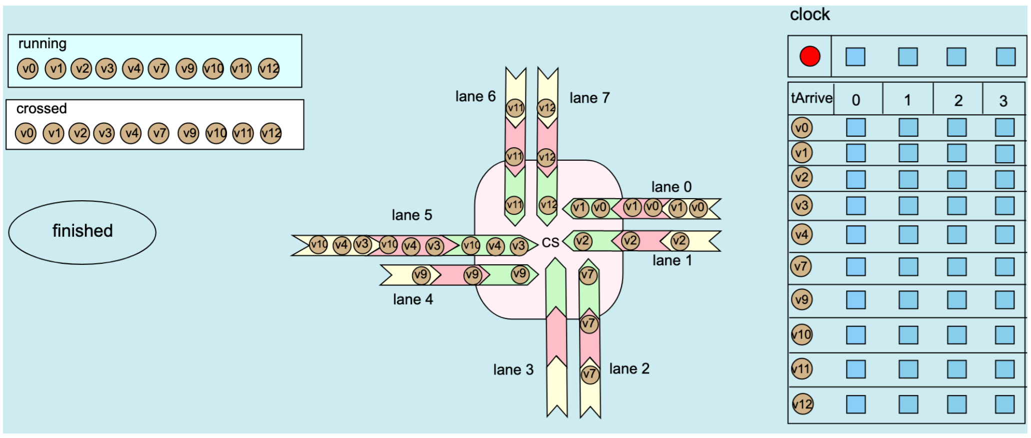


Figure 4: A state picture design for the LJPL protocol (2)

5. Graphical Animation of LJPL Protocol

5.1. Idea

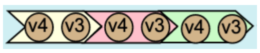
At the beginning of the work, we needed to deal with a problem of how to design a good state picture so that it can display well a composite value of some observable component. At that time, the version of SMGA was only able to visualize observable components whose value is text or designated place. The tool, however, was not able to display specific components inside the composite values of the observable components. For example, considering the following observable component: $(time\colon\ (YY, MM, DD))$, SMGA was not able to display the value "YY", "MM", or "DD". Therefore, we have modified the specification by adding some observable components that do not affect the behavior of the protocol. With the example above, three observable components are added: $(year\colon\ YY)$ $(month\colon\ MM)$ $(day\colon\ DD)$ $(time\colon\ (YY, MM, DD))$. This is the key idea to makes the tool be able to produce good graphical animations for the LJPL protocol in particular and display observable components whose values are composite in general.

It is convenient for users if SMGA supports a functionality that can explicitly visualize specific component inside the composite values of the observable components without adding unneeded observable components. Therefore, we have revised the tool to support the functionality. The key idea is to add # followed by a natural number (start from 0) that represents a position inside the composite value of an observable component. For example, with the following composite value $(time\colon\ (YY, MM, DD(hh, mm, ss)))$, where its third component is also a composite value that consists of hh, mm and ss, we can extract the value mm by the notation *time*#2#1, where 2 denotes the third position of *time*'s value (i.e., $DD(hh, mm, ss)$), and 1 denotes the second position inside DD (i.e., mm). Therefore, users can display the component as a text or a designated place with the new functionality provided by SMGA.

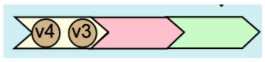
5.2. State Picture Design

In SMGA, designing a good state picture is an important task because it can help humans better perceive the characteristics of the protocol [5]. In our way to formalize the LJPL protocol, each state is expressed as a braced soup of observable components. Multiple similar observable components, such as `v` observable components, are used. `v` observable components contain values whose types are the same, such as the lane ID (`laneID`) and the *status* (`vStat`) of a vehicle. By following the similarity principle of Gestalt [20, 19], they should be put together. Furthermore, the laneID of a vehicle cannot be changed and hence we fix it as a constant text. We then come up with a state picture design for the initial state `init` mentioned in the previous section (shown in Figure 2). A state picture generated from the state picture design is depicted in Figure 3.

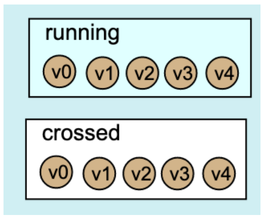
In Figure 2, there are eight arrow shapes representing eight lanes. A lane representation designed is as follows:



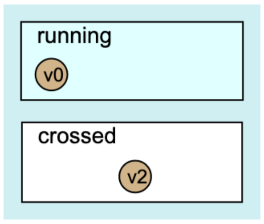
There are three colors: light green, pink, and light yellow that represent three statuses crossing, stopped, and approaching, respectively. For example, the status values of the fourth and fifth vehicles (i.e., v3 and v4) in the following figure are approaching:



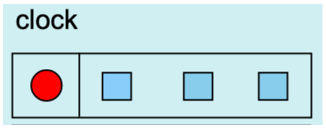
Two status values running and crossed representations used in Figure 2 are as follows:



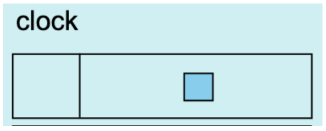
A rectangle whose color is light cyan represents the status running. A rectangle whose color is white represents the status crossed. For example, the status values of the first and third vehicles (i.e., v0 and v2) in the following figure are running and crossed, respectively:



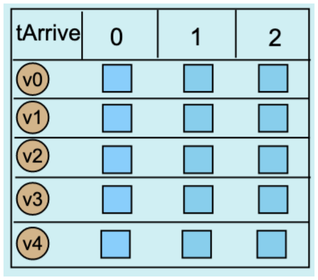
The design of the clock representation used in Figure 2 is as follows:



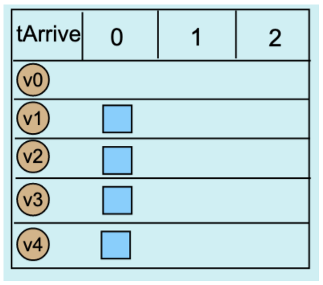
The value of the clock consists of two pieces of information: a natural number and a Boolean value (mentioned in Sect. 4). Three blue squares represent the natural number from 0 to 2. If the value of the natural number is 0, the first blue square is displayed. A red circle represents the Boolean value. If the value is *true*, a red circle is displayed, otherwise, nothing is displayed. For example, when the value of the natural number is 1, and the Boolean value is $false$, those values are displayed as follows:



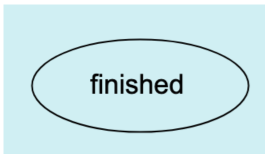
The design of the time arrivals of vehicles representation used in Figure 2 are as follows:



In each horizontal line, three blue squares represent the value of the time arrival (from 0 to 2). If nothing is displayed, the value is ∞. If the value is 0, the first blue square is displayed. For example, the figure below displays the case when the value of the first vehicle is ∞, the values of the four other vehicles are 0:



The design of the gstat representation used in Figure 2 is as follows:



If the value of gstat is `fin`, the circle and text is displayed, otherwise, nothing is displayed.

Figure 5 shows a state picture in which the initial state contains one vehicle in each lane1, lane2, lane4, lane6, and lane7, two vehicles in lane0, and three vehicles in lane5. It indicates that users need to redesign a new state picture since the initial state is changed. We design a flexible state picture such that it can be used when the number of vehicles participating in the protocol is small enough. Figure 7 displays the flexible state picture design, in which each lane can contain up to four vehicles, the value of the natural number of clock, and the value of the time arrival are up to 6.

5.3. Graphical Animation of LJPL Protocol

Figure 6 shows a state sequence for the LJPL protocol based on the state picture design depicted in Figure 5. Six pictures correspond to six consecutive states from State 13 to State 18 in one state sequence randomly generated by Maude. Those pictures follow the rewrite rules mentioned in 4, such as State 17 is the successor of State 16 by the rewrite rule `leave1`. Taking a look at the first picture (State 13) immediately makes us recognize that each of lane0 and lane5 contains two vehicles whose status values are stopped, each of lane6 and lane7 contains one vehicle whose status value is approaching, each of lane1 and lane2 contains one vehicle whose status value is stopped, the values of time arrival of those vehicles are equal to 0 except two vehicles whose status values are running have time arrival ∞. Taking a look at State 13 and State 14 immediately makes us recognize that v12's status changes from approaching to stopped. Taking a look at State 15 to State 17 immediately makes us recognize that v2's status value changes from stopped to crossing and finally to crossed, and v4's status value changes from running to approaching.

Figure 8 shows another state sequence. Three pictures correspond to three consecutive states from State 27 to State 29. Taking a look at State 27 and State 28 makes us immediately recognize that three vehicles can change the status from stopped to crossing at the same time. It is interesting because crossing is regarded as the critical section such that

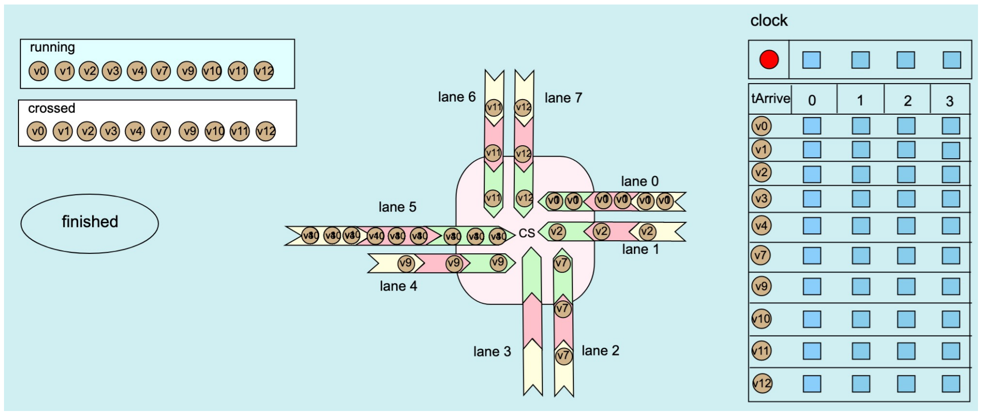
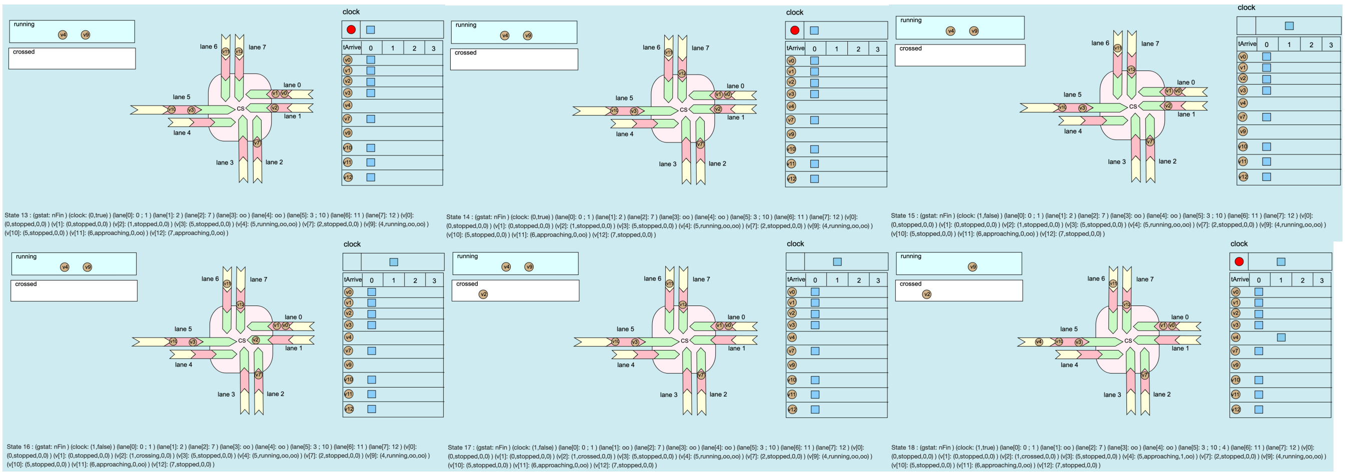


Figure 5: A state picture design for the LJPL protocol (3)



```
State 13 : (gstat: nFin ) (clock: (0,true) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,approaching,0,oo) )
```

```
State 14 : (gstat: nFin ) (clock: (0,true) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

```
State 15 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

```
State 16 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,crossing,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

```
State 17 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: oo ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,crossed,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

```
State 18 : (gstat: nFin ) (clock: (1,true) ) (lane[0]: 0 ; 1 ) (lane[1]: oo ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ; 4 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,crossed,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,approaching,1,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) )
(v[10]: (5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

Figure 6: A state sequence for the LJPL protocol (1)

at most one vehicle should be located in the critical section at the same time. Taking a look at the State 28 makes us immediately recognize a case such that there exist two vehicles running on two different lanes, and their statuses are crossing. It can be explained that two vehicles running on two concurrent lanes (e.g., lane5 and lane2) are allowed to cross the intersection simultaneously.

5.4. Preservation of the Order of Vehicles on each Lane

In graphical animations of the LJPL protocol produced in our conference paper [14], the order of vehicles on each lane in each state picture instance may be different from the actual order of the vehicles on the lane. For example, in State 13 appearing in Figure 6, there are two `v0` and `v1` on lane0. `v1` is the first vehicle and `v0` is the second one on the state picture instance (State 13), while the `lane[0]` observable component has `v0 ; v1` as its value, meaning that `v0` is the first vehicle and `v1` is the second one. This difference is not good because human users may mis-understand something about State 13 by looking at the state picture. This is because the version of SMGA available when our conference paper [14] was written required us to fix the position for each vehicle on each lane when the vehicle status is one of the three statuses (approaching, stopped and crossing). Therefore, `v1` is always in front of `v0` in the state picture design used in the conference version even though `v0` is actually in front of `v1` when the statuses of both `v0` and `v1` are one of the three statuses.

One possible way to solve the situation is to use text display (see Sect. 2) to visualize queues of `lane` observable components. However, it is impossible to extract each vehicles' statuses from the queues because the queues only consists of vehicle IDs. For example, when the `lane[0]` observable component has `v0 ; v1` as its value, the text `v0 ; v1` is displayed. Because vehicles statuses are one piece of essential information of the protocol, this approach is not good enough.

We took a different approach. We have revised SMGA so that the tool can use mark display (see Sect. 2) to visualize queues of `lane` observable components so as to visualize vehicles statuses effectively. When a vehicle is in one of the three statuses, its ID is always displayed at a fixed

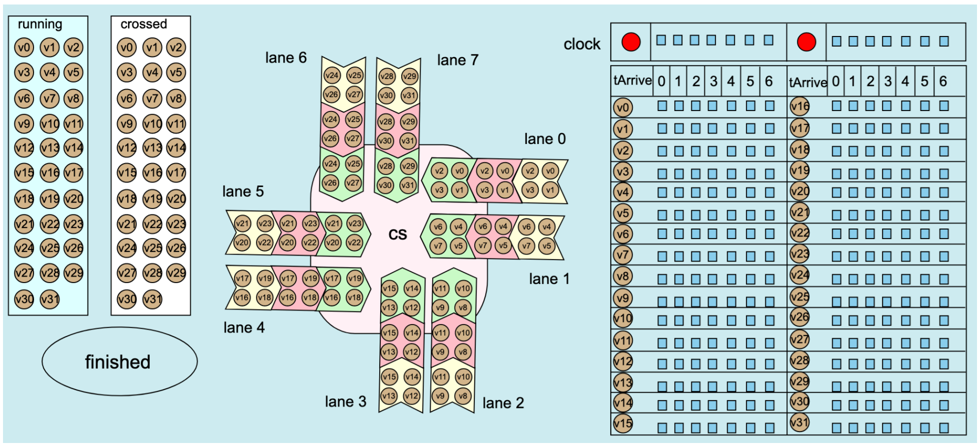
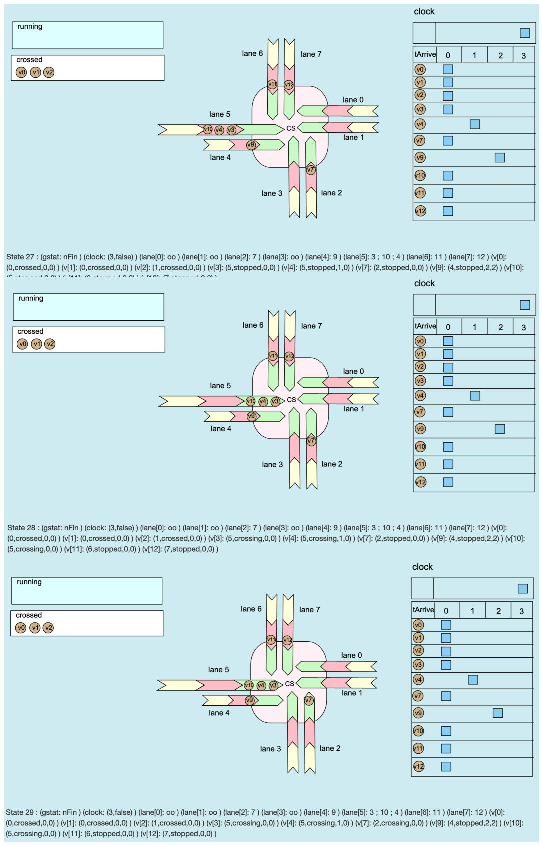


Figure 7: A flexible state picture design for the LJPL protocol (1)



```
State 27 : (gstat: nFin ) (clock: (3,false) ) (lane[0]: oo ) (lane[1]: oo ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: 9 ) (lane[5]: 3 ; 10 ; 4 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,crossed,0,0) ) (v[1]: (0,crossed,0,0) ) (v[2]: (1,crossed,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,stopped,1,0) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,stopped,2,2) ) (v[10]:
(5,stopped,0,0) ) (v[11]: (6,stopped,0,0) ) (v[12]: (7,stopped,0,0) )
```

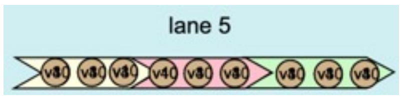
```
State 28 : (gstat: nFin ) (clock: (3,false) ) (lane[0]: oo ) (lane[1]: oo ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: 9 ) (lane[5]: 3 ; 10 ; 4 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,crossed,0,0) ) (v[1]: (0,crossed,0,0) ) (v[2]: (1,crossed,0,0) ) (v[3]: (5,crossing,0,0) ) (v[4]: (5,crossing,1,0) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,stopped,2,2) ) (v[10]:
(5,crossing,0,0) ) (v[11]: (6,stopped,0,0) ) (v[12]: (7,stopped,0,0) )
```

```
State 29 : (gstat: nFin ) (clock: (3,false) ) (lane[0]: oo ) (lane[1]: oo ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: 9 ) (lane[5]: 3 ; 10 ; 4 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,crossed,0,0) ) (v[1]: (0,crossed,0,0) ) (v[2]: (1,crossed,0,0) ) (v[3]: (5,crossing,0,0) ) (v[4]: (5,crossing,1,0) ) (v[7]: (2,crossing,0,0) ) (v[9]: (4,stopped,2,2) ) (v[10]:
(5,crossing,0,0) ) (v[11]: (6,stopped,0,0) ) (v[12]: (7,stopped,0,0) )
```

Figure 8: A state sequence for the LJPL protocol (2)

location of the arrow shape for the status in the previous approach. Thus, state pictures may not respect the order in which there are vehicles whose statuses are one of the three statuses (approaching, stopped and crossing) on a lane. On the other hand, in the current approach taken in the present paper, we allows the ID of such a vehicle to be displayed at any possible location of the arrow shape depending on the value (queue) of the lane observable component concerned.

For example, the design of the arrow shape for lane 5 is as follows:



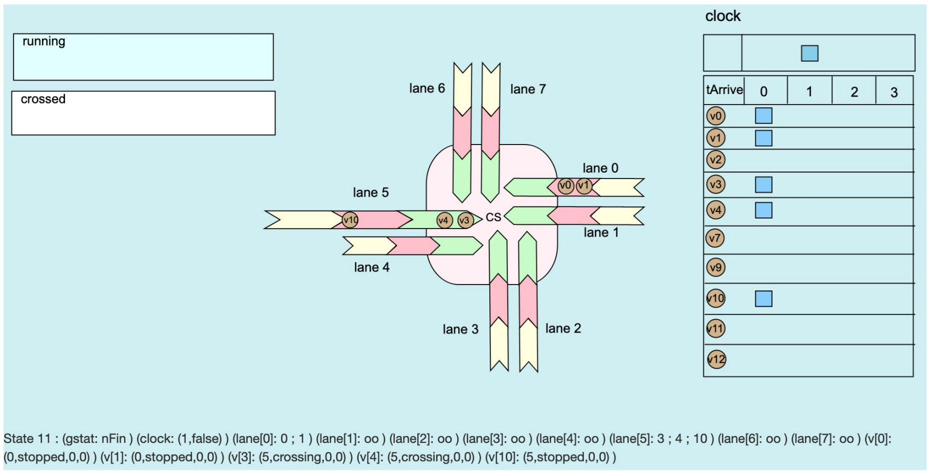
We suppose that there are three vehicles `v3`, `v4` and `v10` on lane 5. The design is made so that each of the three vehicles can be displayed at any of the three possible positions on each arrow sub-shape (note that there are three arrow sub-shapes on the lane).

When the value (queue) of the `lane[`i`]` observable component is displayed, SMGA checks the status of each vehicle j in the queue by looking at the second component ($vstat$) of the `v[`j`]` observable component. For example, when there are three vehicles `v3`, `v4` and `v10` on lane 5 such that `v3` and `v4` are crossing and `v10` is stopped, they are displayed as shown in Figure 9. The state picture respects the order of the three vehicles `v3`, `v4` and `v10` on lane 5.

Figure 10 shows a sequence of state pictures produced by the latest version of SMGA that corresponds to the sequence of state pictures shown in Figure 8. Each state picture appearing in Figure 10 preserves the order of vehicles on each lane, while each state picture appearing in Figure 8 does not necessarily preserve the order.

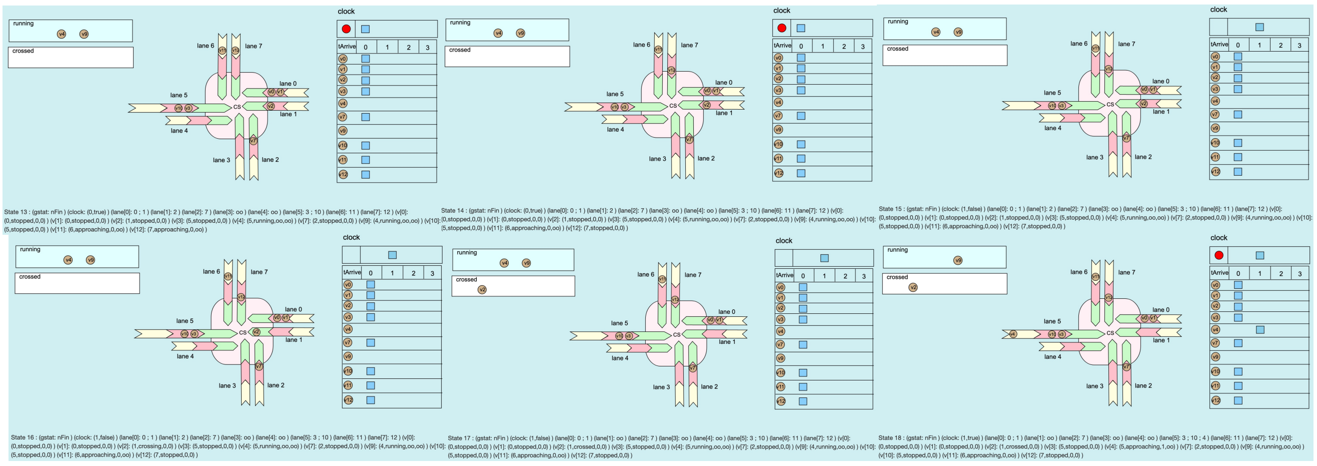
5.5. Visualization of Conflict and Concurrent Lanes

For each of the eight lanes, there are four conflict and three concurrent lanes. In graphical animations of the LJPL protocol produced in our conference paper [14], any information on conflict and concurrent lanes is displayed. Information on conflict and concurrent lanes is not very static because each lane has different conflict and concurrent lanes. Furthermore, information on conflict and concurrent lanes is not stored in any observable components. Thus, such information cannot be handled as any other information, such as the status of each vehicle and the lane information. Be-



State 11 : (gstat: nFin) (clock: (1,false)) (lane[0]: 0 ; 1) (lane[1]: oo) (lane[2]: oo) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 4 ; 10) (lane[6]: oo) (lane[7]: oo) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[3]: (5,crossing,0,0)) (v[4]: (5,crossing,0,0)) (v[10]: (5,stopped,0,0))

Figure 9: A state picture for the LJPL protocol (2)



State 13 : (gstat: nFin) (clock: (0,true)) (lane[0]: 0 ; 1) (lane[1]: 2) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,stopped,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,running,oo,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,approaching,0,oo))

State 14 : (gstat: nFin) (clock: (0,true)) (lane[0]: 0 ; 1) (lane[1]: 2) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,stopped,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,running,oo,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,stopped,0,0))

State 15 : (gstat: nFin) (clock: (1,false)) (lane[0]: 0 ; 1) (lane[1]: 2) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,stopped,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,running,oo,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,stopped,0,0))

State 16 : (gstat: nFin) (clock: (1,false)) (lane[0]: 0 ; 1) (lane[1]: 2) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,crossing,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,running,oo,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,stopped,0,0))

State 17 : (gstat: nFin) (clock: (1,false)) (lane[0]: 0 ; 1) (lane[1]: oo) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,crossed,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,running,oo,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,stopped,0,0))

State 18 : (gstat: nFin) (clock: (1,true)) (lane[0]: 0 ; 1) (lane[1]: oo) (lane[2]: 7) (lane[3]: oo) (lane[4]: oo) (lane[5]: 3 ; 10 ; 4) (lane[6]: 11) (lane[7]: 12) (v[0]: (0,stopped,0,0)) (v[1]: (0,stopped,0,0)) (v[2]: (1,crossed,0,0)) (v[3]: (5,stopped,0,0)) (v[4]: (5,approaching,1,oo)) (v[7]: (2,stopped,0,0)) (v[9]: (4,running,oo,oo)) (v[10]: (5,stopped,0,0)) (v[11]: (6,approaching,0,oo)) (v[12]: (7,stopped,0,0))

Figure 10: A state sequence for the LJPL protocol (3)

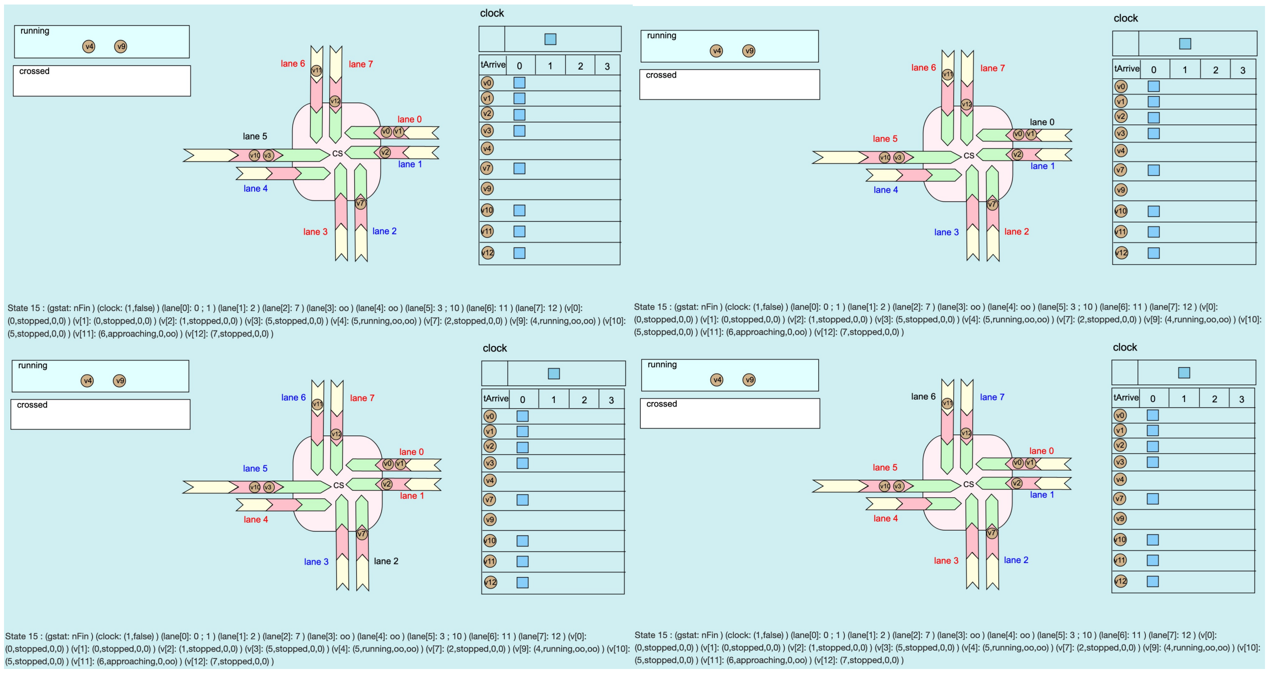
cause information on conflict and concurrent lanes is crucial for the LJPL protocol, it should be visualized.

Our approach to visualization of information on conflict and concurrent lanes is as follows. We make each text "lane i" for $i = 0, 1, \ldots, 7$ on each state picture clickable. Initially, each text "lane i" is displayed in black. When "lane i" is clicked, its four conflict lane texts become red and its three concurrent lane texts become blue, while "lane i" is kept black. For example, when "lane 5" is clicked, each "lane j" for $j = 0, 3, 6, 7$ becomes red and each "lane k" for $k = 1, 2, 4$ becomes blue. Even while playing a graphical animation, each "lane i" can be clicked. Figure 11 shows four state pictures in which "lane 5", "lane 0", "lane 2" and "lane 6", respectively. The functionality that visualizes the conflict and concurrent lanes of "lane i" by clicking "lane i" is called the conflict/concurrent lane interaction functionality.

Let us note that there are four state pictures in Figure 11 but they represent one state (State 15). State 15 is an example in which a deadlock situation occurs, when no vehicle will cross the intersection. It is reported by Moe et al. [2] that the original LJPL protocol does not enjoy the deadlock freedom property. When the lead arrival times of the two top vehicles on two conflict lanes are exactly the same, the original LJPL protocol cannot select one of the two vehicles. Moe et al. [2] propose that when that is the case, one vehicle whose lane ID is less than the other vehicle's lane ID is selected. We use the revised protocol by Moe et al. [2] in the following sections.

6. Confirmation of Guessed Characteristics with Maude

We first guess four characteristics by observing graphical animations of the LJPL protocol and confirm them with the Maude search command. We then describe one seeming characteristic by observing graphical animations of the LJPL protocol. The characteristic is refuted by the Maude search command. We revise the characteristic and confirm it by the Maude search command. Let `init` be the initial state



```
State 15 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) ) (v[10]:
(5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )

State 15 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) ) (v[10]:
(5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )

State 15 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) ) (v[10]:
(5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )

State 15 : (gstat: nFin ) (clock: (1,false) ) (lane[0]: 0 ; 1 ) (lane[1]: 2 ) (lane[2]: 7 ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ; 10 ) (lane[6]: 11 ) (lane[7]: 12 ) (v[0]:
(0,stopped,0,0) ) (v[1]: (0,stopped,0,0) ) (v[2]: (1,stopped,0,0) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) ) (v[7]: (2,stopped,0,0) ) (v[9]: (4,running,oo,oo) ) (v[10]:
(5,stopped,0,0) ) (v[11]: (6,approaching,0,oo) ) (v[12]: (7,stopped,0,0) )
```

Figure 11: Four state pictures in which lanes 5, 0, 2, 6 are clicked, respectively

defined in Sect. 4.

6.1. Four Characteristics Guessed and Confirmed

Observing some graphical animations, we first see that the status of a vehicle sometimes does not change when the value of clock changes (shown at State 15 in Figure 6). Carefully focusing on the value of clock, we guess that when the Boolean value of clock is false, the arrival time of any vehicle cannot be greater than the first value of clock. The characteristic can be confirmed by the Maude search command as follows:

```
search [1] in RIMUTEX : init =>*
{(clock: T,false) (v[VI]: LI,VS,T1,T2) OCs}
such that
T1 >= T and T1 =/= oo .
```

The search command above tries to find a reachable state in which the arrival time `T1` (that is not ∞) of a vehicle `VI` is greater than or equal to the first value `T` of clock. Maude does not find any reachable state that satisfies the condition from the state `init`. Therefore, the guessed characteristic is confirmed with the initial state `init`.

Observing some graphical animations, we guess that if the first value of clock is equal to the arrival time of a vehicle, the status of the vehicle is not running. The characteristics can be confirmed by Maude search command as follows:

```
search [1] in RIMUTEX : init =>*
{(clock: T,B) (v[VI]: LI,VS,T1,T2) OCs}
such that
T1 == T and VS == running .
```

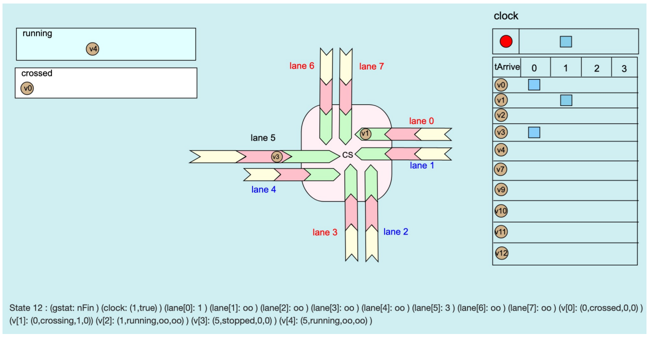
The search command above tries to find a reachable state in which the arrival time `T1` of a vehicle `VI` is equal to the first value `T` of clock and the status of the vehicle is running. Maude does not find any reachable state that satisfies the condition from the state `init`. Consequently, the guessed characteristic is confirmed with the initial state `init`.

Observing some graphical animations with the conflict/concurrent lane interaction functionality, we recognize that if a vehicle is the top of lane `LI1` and its status is crossing, then another vehicle that is the top of lane `LI2` that is a conflict one of lane `LI1` is never crossing. The charismatic can be confirmed by the Maude search command as follows:

```
search [1] in RIMUTEX : init =>*
{(lane[LI1]: VI1 ; Q1) (v[VI1]: LI1,crossing,T11,T12)
 (lane[LI2]: VI2 ; Q2) (v[VI2]: LI2,crossing,T12,T22) OCs}
such that
areConflict(LI1,LI2) = true /\ LI1 =/= LI2 .
```

The search command above tries to find a reachable state in which two lanes `LI1` and `LI2` are conflict, two vehicles `VI1` and `VI2` are the top of the two lanes, respectively, and the two vehicles' statuses are crossing. Maude does not find any reachable state that satisfies the condition from `init`. Therefore, the guessed characteristic is confirmed with the initial state `init`.

Observing some graphical animations, we carefully focus on concurrent lanes. Let us note that there are three concurrent lanes for each lane. We recognize that there are at most two concurrent lanes on which vehicles are crossing simultaneously. The charismatic can be confirmed by Maude search command as follows:



```
State 12 : (gstat: nFin ) (clock: (1,true) ) (lane[0]: 1 ) (lane[1]: oo ) (lane[2]: oo ) (lane[3]: oo ) (lane[4]: oo ) (lane[5]: 3 ) (lane[6]: oo ) (lane[7]: oo ) (v[0]: (0,crossed,0,0) )
(v[1]: (0,crossing,1,0)) (v[2]: (1,running,oo,oo) ) (v[3]: (5,stopped,0,0) ) (v[4]: (5,running,oo,oo) )
```

Figure 12: A state picture in which the seeming characteristic is broken

```
search [1] in RIMUTEX : init =>*
{(lane[L1]: v1 ; Q1) (lane[L2]: v2 ; Q2)
 (lane[L3]: v3 ; Q2)
 (v[V1]: L1,crossing,T11,T12)
 (v[V2]: L2,crossing,T21,T22)
 (v[V3]: L3,crossing,T31,T32) OCs}
such that
areConcur(L1,L2) /\ areConcur(L1,L3) /\
areConcur(L2,L3) /\ (L1 =/= L2) /\ (L1 =/= L3)
/\ (L2 =/= L3) .
```

The search command above tries to find a reachable state in which three different lanes `L1`, `L2` and `L3` are concurrent, three vehicles `V1`, `V2` and `V3` are the top of the three lanes, respectively, and the three vehicles are crossing. Maude does not find any reachable state that satisfies the condition from `init`. Therefore, the guessed characteristic is confirmed with the initial state `init`.

6.2. One Seeming Characteristic and its Revision

By using one tip called CCT-2 (By concentrating on two different-kind observable components, we may find a relation between them, from which we may conjecture some characteristics.) [6], we carefully focus on the two top vehicles `VI1` and `VI2` of two conflict lanes `LI1` and `LI2`, the statuses of `VI1` and `VI2` and the arrival times `T11` and `T21` of `VI1` and `VI2`. We then guess that if `VI1` is crossing, then `T11` is less than or equal to `T21`. We tried to confirm the characteristic with the following Maude search command:

```
search [1] in RIMUTEX : init =>*
{(lane[LI1]: VI1 ; Q1) (v[VI1]: LI1,crossing,T11,T12)
 (lane[LI2]: VI2 ; Q2) (v[VI2]: LI2,VS2,T21,T22) OCs}
such that
areConflict(LI1,LI2) = true /\
LI1 =/= LI2 /\ VI1 =/= VI2 /\ T11 > T21 .
```

Maude found a counterexample of the guessed characteristic, although it is seemingly correct. Figure 12 shows a state picture of the counterexample. Let us take a look at the vehicles `v1` on lane 0 and `v5` on lane 5. The two lanes are conflict and the two vehicles are the top of the lanes as shown in Figure 12. `v1` is crossing and `v5` is stopped, while the `v1`'s arrival time is 1 and the `v5`'s arrival time is 0. `v1` is the top of lane 0 and seems to be the lead vehicle of lane 0, but `v1` is not the lead vehicle of lane 0. `v0` that is crossed was the lead vehicle of lane 0 because both its arrival time and lead arrival time (the third and forth components of `v[v0]` observable component) are 0 and `v1` has been following `v0` when crossing the intersection. This is because the lead arrival time of `v1` is 0 that is the same as the arrival time of `v0`.

To guess the characteristic, we should have considered the lead arrival time instead of the arrival time of each vehicle concerned. We revise the characteristic as follows: if there are the top vehicles `VI1` and `VI2` of two conflict lanes `LI1` and `LI2` such that `VI1` is crossing, the lead arrival time `T12` of `VI1` is less than or equal to the lead arrival time `T22` of `VI2`. The revised characteristic can be confirmed by the following Maude search command:

```
search [1] in RIMUTEX : init =>*
{(lane[LI1]: VI1 ; Q1) (v[VI1]: LI1,crossing,T11,T12)
 (lane[LI2]: VI2 ; Q2) (v[VI2]: LI2,VS2,T21,T22) OCs}
such that
areConflict(LI1,LI2) = true /\
LI1 =/= LI2 /\ VI1 =/= VI2 /\ T12 > T22 .
```

Maude does not find any counterexamples.

7. Lessons Learned

Through the case study with the LJPL protocol, we obtain several lessons on how to design a good state picture so that we can conjecture some non-trivial characteristics, especially when there are some observable components with composite values. The lessons learned can be summarized as follows:

- When an observable component has a composite value, which consists of more than one component value inside, we need to carefully select which component values to visualize. For example, the second and the third component values (i.e., the *status* and the time arrival) of the vehicle observable component are selected while the fourth component value (i.e., the time arrival of the lead) of the vehicle observable component is not used in our design.
- If a value of an observable component does not change, it should be expressed at a fixed label, such as laneID of each vehicle observable component.
- If there exist observable components that have values whose types are the same, we should design and display the values together in one designated place.
- If there exist observable components that have a natural number as their values and the values are small enough, the values should be visually expressed nearby together so that we can see them simultaneously and compare them instantaneously. For example, the first value of clock (i.e., a natural number) and the time arrival of each vehicle have been visualized in our design.

8. Related Work

Bui and Ogata [4] have graphically animated a mutual exclusion distributed protocol called Suzuki-Kasami with SMGA. The protocol contains the network component used to exchange messages. They have realized the messages that have been put into the network and deleted from the network are crucial information so that they have revised SMGA to be able to display those messages. Their solution is to prepare two places for those messages. Some guessed characteristics are confirmed as invariant properties of the protocol with Maude. The authors have summed up their experiences as tips to help human users design a state picture for distributed protocol. This research and ours can share the working flow, but we cannot apply the technical content or tips to our work. One such reason is that the behavior of distributed protocol and autonomous vehicle intersection control protocol cannot share each other.

A mutual exclusion protocol called Qlock has been conducted by May Thu Aung et al. [3]. Some properties of the protocol have been conjectured by observing graphical animation. The properties also have model checked with Maude and theorem proved with CafeOBJ [8]. One piece of our future work is to theorem prove the characteristics guessed in this paper with CafeOBJ.

María Alpuente et al. [13] have proposed a methodology to check whether a Maude program is correct or not via logical assertions based on rewriting logic theories. They also have developed a prototype tool that is an implementation of that methodology. One functionality of the tool is to visualize the possible trace slides (as state sequences in our paper) to help users identify the cause of the error. Human users can observe the specific states corresponding to their rewriting rules by selecting them from a given initial state. In case that many possible rewriting rules may appear, the visualization is looked like a graph or a tree in which the states (displayed as text) are nodes. This visualization approach can be applied to our work, although its purpose is different than ours. One piece of our work is to compare those approaches together.

SMGA can be regarded as an integration of formal methods and visualization. We introduce two recent studies on an integration of formal methods and visualization. One [9] is a study on visualization of what is done inside by Vampire [11], an automated first-order logic theorem prover, and the other is a study on visualization of the structural operational semantics of a simple imperative programming language [17]. Although automated theorem provers are attractive because they may automatically prove theorems, they cannot truly fully automatically prove all possible theorems. Proof attempts may fail. If that is the case, human users need to comprehend why the proof attempts fail and need to change the format of input logical formulas and/or some internal proof strategies. It is very difficult for non-expert users and at least non-trivial for expert users to really comprehend why the proof attempts fail because it is necessary to understand what is done inside by an automated theorem prover, such as Vampire. Gleiss, et al. have then developed SatVis, a tool to visualize what is done inside by Vampire. Students and even programmers should learn semantics of programming languages so as to understand programming languages better, which may make it possible for them to write better programs. However, it is hard for students to learn semantics of programming languages. Perhác and Zuzana Bilanová have then developed an interactive tool for visualization of the structural operational semantics of a simple imperative programming language. SMGA partially shares the motivation of the first study [9]. This is because the main purpose of SMGA is to help human users conjecture lemmas needed for interactive theorem proving through graphical animations of state machines concerned. Nothing special is directly shared by SMGA and the second study [17] except an integration of formal methods and visualization. However, several formal semantics of programing languages have been described in the $\mathbb{K}$ framework [18], where $\mathbb{K}$ has been implemented in Maude. SMGA basically graphical animates state machines specified in Maude. Therefore, it would be possible to integrate SMGA and the $\mathbb{K}$ framework so that formal semantics of programming languages can be visualized.

9. Conclusion

We have described graphical animations of the Lim-Jeong-Park-Lee autonomous intersection control protocol with SMGA in which the composite data are explicitly visually displayed. Two state picture designs that deal with initial states in two different ways have been created, and a flexible state picture design has been proposed so that all initial states can be handled provided that the number of vehicles is less than or equal to a given number. Some characteristics are guessed by observing graphical animations based on our design to demonstrate that graphical animation could help humans visually perceive the characteristics of the protocol. Those characteristics are confirmed by model checking. We have summarized our experiences as some tips on how to design a good state picture for autonomous vehicle intersection control protocol. One future direction is to apply our work to other self-driving vehicle protocols, such as a merging protocol [1]. Another future work is to integrate SMGA into Maude so that the tool can use some functionalities of Maude, such as pattern matching and generating a state sequence of state on the fly.

Acknowledgment

The authors would like to thank the anonymous reviewers who carefully read an earlier version of the paper and gave them valuable comments without which they were not able to complete the present paper.

References

[1] Aoki, S., Rajkumar, R., 2017. A merging protocol for self-driving vehicles, in: ICCPS 2017, pp. 219–228. doi:10.1145/3055004.3055028.
[2] Aung, M.N., Phyo, Y., Ogata, K., 2019. Formal specification and model checking of the Lim-Jeong-Park-Lee autonomous vehicle in-

tersection control protocol, in: SEKE 2019, pp. 159–208. doi:10.18293/SEKE2019-021.

[3] Aung, M.T., Nguyen, T.T.T., Ogata, K., 2018. Guessing, model checking and theorem proving of state machine properties – a case study on Qlock. IJSECS 4, 1–18. doi:10.15282/ijsecs.4.2.2018.1.0045.

[4] Bui, D.D., Ogata, K., 2019. Graphical animations of the Suzuki-Kasami distributed mutual exclusion protocol. JVLC 2019, 105–115. doi:10.1007/978-3-319-90104-6_1.

[5] Bui, D.D., Ogata, K., 2020. Better state pictures facilitating state machine characteristic conjecture, in: DMSVIVA 2020, pp. 7–12. doi:10.18293/DMSVIVA20-007.

[6] Bui, D.D., Ogata, K., 2021. Better state pictures facilitating state machine characteristic conjecture. Multimedia Tools and Applications doi:10.1007/s11042-021-10992-z.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (Eds.), 2007. All About Maude. volume 4350 of *LNCS*. Springer. doi:10.1007/978-3-540-71999-1.

[8] Diaconescu, R., Futatsugi, K., 1998. CafeOBJ Report. World Scientific.

[9] Gleiss, B., Kovács, L., Schnedlitz, L., 2019. Interactive visualization of saturation attempts in Vampire, in: IFM 2019, Springer. pp. 504–513. doi:10.1007/978-3-030-34968-4_28.

[10] K. W. Brodlie, et al. (Ed.), 1992. Scientific Visualization: Techniques and Applications. Springer. doi:10.1007/978-3-642-76942-9.

[11] Kovács, L., Voronkov, A., 2013. First-order theorem proving and Vampire, in: CAV 2013, Springer. pp. 1–35. doi:10.1007/978-3-642-39799-8_1.

[12] Lim, J., Jeong, Y., Park, D., Lee, H., 2018. An efficient distributed mutual exclusion algorithm for intersection traffic control. J. Supercomput. 74, 1090–1107. doi:10.1007/s11227-016-1799-3.

[13] M. Alpuente, et al., 2016. Debugging Maude programs via runtime assertion checking and trace slicing. J. Log. Algebraic Methods Program. 85, 707–736. doi:10.1016/j.jlamp.2016.03.001.

[14] Myint, W.H.H., Bui, D.D., Tran, D.D., Ogata, K., 2021. Graphical animations of the Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol, in: DMSVIVA 2021, pp. 22–28. doi:10.18293/DMSVIVA2021-004.

[15] Nguyen, T.T.T., Ogata, K., 2017a. Graphical animations of state machines, in: 15th DASC, pp. 604–611. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.107.

[16] Nguyen, T.T.T., Ogata, K., 2017b. Graphically perceiving characteristics of the MCS lock and model checking them, in: 7th SOFL+MSVL, pp. 3–23. doi:10.1007/978-3-319-90104-6_1.

[17] Perhác, J., Bilanová, Z., 2020. Another tool for structural operational semantics visualization of simple imperative language, in: ICETA 2020, IEEE. pp. 513–518. doi:10.1109/ICETA51985.2020.9379205.

[18] Rosu, G., 2017. 𝕂: A semantic framework for programming languages and formal analysis tools, in: Dependable Software Systems Engineering. IOS Press. volume 50, pp. 186–206. doi:10.3233/978-1-61499-810-5-186.

[19] Todorovic, D., 2008. Gestalt principles. Scholarpedia 3, 5345. doi:10.4249/scholarpedia.5345.

[20] Ware, C., 2004. Information Visualization: Perception for Design. MKP Inc.