Graphical Animations of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol

Win Hlaing Hlaing Myint, Dang Duy Bui, Duong Dinh Tran, Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {winhlainghlaingmyint,bddang,duongtd,ogata}@jaist.ac.jp

***Abstract*—The main goal of SMGA is to help human users be able to perceive (non-trivial) characteristics. The Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol has been graphically animated with SMGA to demonstrate that SMGA can be applied to the wider class of systems. We have revised SMGA so as to handle composite data that are used in the protocol. 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. 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 make a state picture design when composite data are used**

***Keywords*-graphical animation; LJPL protocol; SMGA; state machine; state picture design**

I. Introduction

SMGA [1] 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 [2]. 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.

The Lim-Jeong-Park-Lee autonomous intersection control protocol (the LJPL protocol) has been graphically animated to demonstrate the potential of SMGA that can be applied to wider classes of systems/protocols. First, we need to carefully make a state picture design because it is a core task of the tool [3]. The specification of the protocol, however, contains some observable components whose values are composite (over one component value inside). One possible way to visualize such a composite value is to display each component value of the composite value. We, however, revise the tool so that users can design a state picture to be able to display the composite data explicitly. Because the LJPL protocol has multiple different initial states even if the number of vehicles is fixed, we make a flexiable state picture template that can be used for different 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. After that, some characteristics are guessed by human users based on some graphical animations of the LJPL protocol and confirmed with Maude [4]. Some lessons learned are summarized as tips on how to make a good state picture design for state machine that use observable components whose values are composite

Bui and Ogata [5] have revised SMGA to visualize the network components. This work and ours can share working flow but we cannot apply those technique as they are to ours. One reason is that the behavior of a distributed mutual exclusion protocol and the LJPL protocol cannot share each other. María Alpuente, et al. [6] have proposed a methodology and implemented a prototype tool to check whether a Maude program is correct or not via logical assertions based on rewriting logic theories. The tool is to visualize the possible trace slides (as state sequences in our paper) to help users identify the cause of the error. In case that many possible rewriting rules are used, the visualization looks 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.

We suppose the readers are familiar with state machines and Maude [4] to some extent. The formal specification of the LJPL protocol used in the paper is followed by the work [8] in which they have proposed some modifications (i.e., two more statuses $\mathrm{running}$ and $\mathrm{approaching}$ are added as each vehicle's $status$). Please refer to the paper [8] in detail.

II. Specification of LJPL Protocol in Maude

In this paper, a state is expressed as a soup of observable components. Let b be a Boolean value, q a queue of vehicle IDs (i.e., a queue of natural numbers). Let vid, lid, t, lt be natural numbers, where vid and lid represent a vehicle ID and a lane ID, respectively, while t and lt represent the time.

This work was partially supported by FY2020 grant-in-aid for new technology research activities at universities (SHIBUYA SCIENCE CULTURE AND SPORTS FOUNDATION)
DOI reference number: 10.18293/DMSVIVA2021-004

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 increased by the time. 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 counterpart b such that t only can increment when b is true. That is, whenever b is true, t can increment, 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` : $gstat$) - 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 `;` constructs the queue. 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.

```
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) .
```

where LIi for $i = 1, \ldots, 4$ are Maude variables of natural numbers, VIi & Tj for $i = 1, \ldots, 4$ & $j = 11, 12, \ldots, 41, 42$ are Maude variables of natural numbers & ∞, VSi for $i = 1, \ldots, 4$ are Maude variables of queues, VSti for $i = 1, \ldots, 4$ are Maude variables of vehicle statuses, and OCsi 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

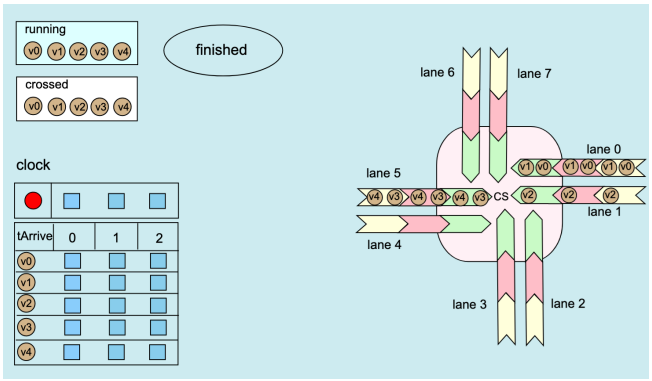


Figure 1. A state picture design for the LJPL protocol (1)

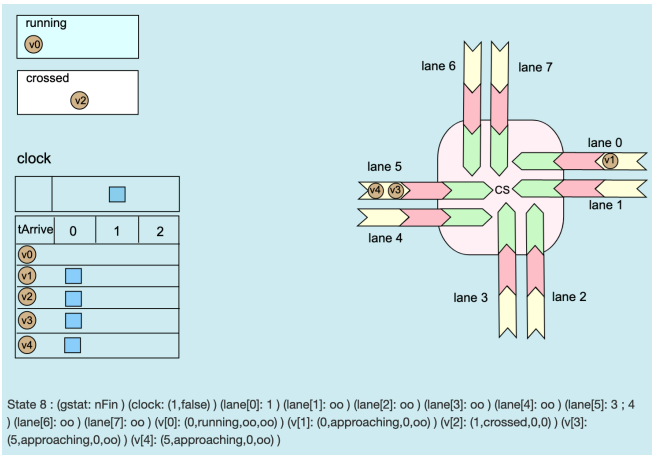


Figure 2. A state picture for the LJPL protocol (1)

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. I.

III. Graphical Animation of LJPL Protocol

A. Idea

At the beginning of the work, the first author of the presented paper needs 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 could only visualize observable components whose value is text or designated place. The tool, however, cannot display specific component inside the composite values of the observable components. For example, considering the following observable component: ($time$: (YY, MM, DD)), SMGA cannot display the value "YY", "MM", or "DD". Therefore, the first author has 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: ($year$: YY) ($month$: MM) (day: DD) ($time$: (YY, MM, DD)). This

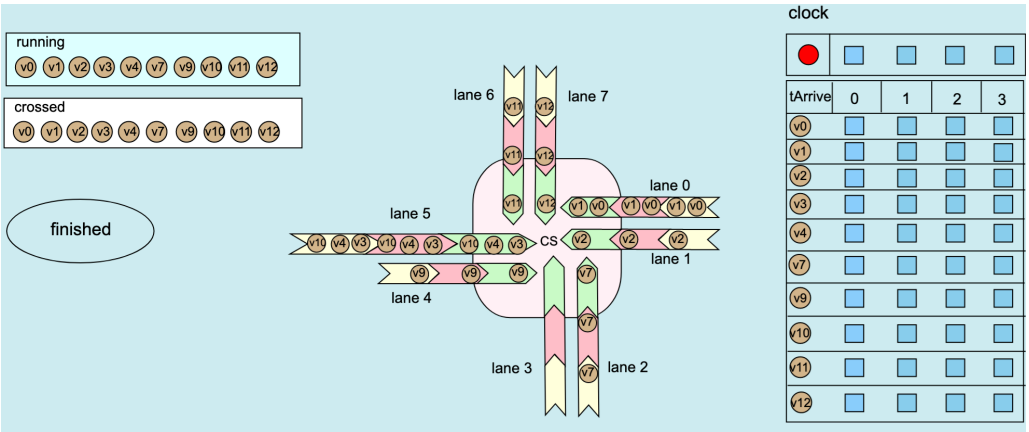


Figure 3. A state picture design for the LJPL protocol (2)

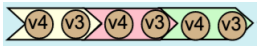
is the key idea to makes the tool be able to produce good graphical animations for the LJPL protocol in particular and display observable components whose values are composite in general.

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, the second author of the presented paper has revised the tool to support that 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 ($time$: $(YY, MM, DD(hh, mm, ss))$), 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 as SMGA provides.

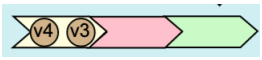
B. State picture design

In SMGA, designing a good state picture is an important task because it can help humans better perceive the characteristics of the protocol [9]. In the LJPL specification, some observable components contain the same values of some information inside such as the lane ID (`laneID`), the $status$ (`vStat`) of a vehicle. By following the similarity principle of Gestalt [10], [11], they should be put together. Furthermore, the laneID of a vehicle cannot be changed, hence, we fix it as a constant text. After that, we come up with a state picture design for the initial state `init` mentioned in the previous section (shown in Fig. 1). A state picture generated from the state picture design is depicted in Fig. 2.

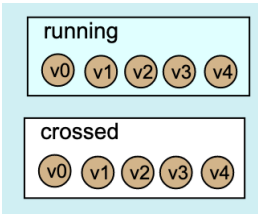
In Fig. 1, there are eight arrow shapes representing eight lanes. A lane representation designed is as follows:



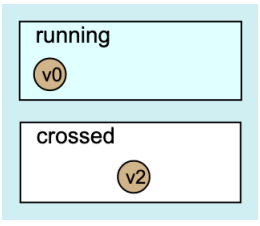
There are three colors: light green, pink, and light yellow that represent three statuses crossing, stopped, and approaching, respectively. For example, the status values of the fourth and fifth vehicles (i.e., v3 and v4) in the following figure are approaching:



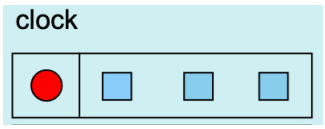
Two status values running and crossed representations used in Fig 1 are as follows:



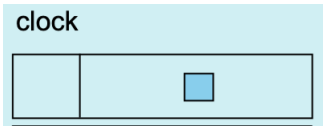
A rectangle whose color is light cyan represents the status running. A rectangle whose color is white represents the status crossed. For example, the status values of the first and third vehicles (i.e., v0 and v2) in the following figure are running and crossed, respectively:



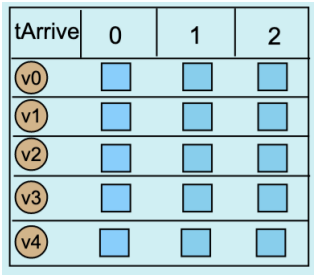
The design of the clock representation used in Fig. 1 is as follows:



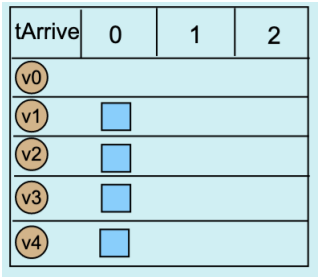
The value of the clock consists of two pieces of information: a natural number and a Boolean value (mentioned in Sect. II). Three blue squares represent the natural number from 0 to 2. If the value of the natural number is 0, the first blue square is displayed. A red circle represents the Boolean value. 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:



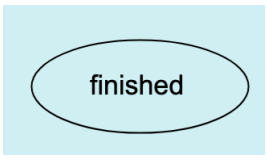
The design of the time arrivals of vehicles representation used in Fig 1 are as follows:



In each 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 Fig. 1 is as follows:



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

Fig. 3 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. Fig. 5 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.

C. Graphical animation of LJPL protocol

Fig. 4 shows a state sequence for the LJPL protocol based on the state picture design depicted in Fig. 3. 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 II. For example, State 17 is the successor of State 16 by the rewrite rule `leave1`. Taking look at the first picture (State 13) immediately makes us recognize that each of lane0 and lane5 contains two vehicles whose status are stopped, each of lane6 and lane7 contains one vehicle whose status is approaching, each of lane1 and lane2 contains one vehicle whose status is stopped, the values of time arrival of those vehicles are equal to 0 except two vehicles whose status are running have time arrival ∞. Taking look at State 13 and State 14 immediately makes us recognize that v12's status changes from approaching to stopped. Taking look at State 15 to State 17 immediately makes us recognize that v2's status changes from stopped to crossing and finally to crossed, and v4's status changes from running to approaching.

Fig. 6 shows another state sequence. Three pictures correspond to three consecutive states from State 27 to State 29. Taking 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 at most one vehicle should be located in the critical section at the same time. Taking look at the State 28 makes us immediately recognize a case that exists 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.

IV. Confirmation of Guessed Characteristics with Maude

Observing the 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 Fig. 4). Carefully focusing on the value of clock, we guess that when the Boolean value of clock is false, the time arrival value of any

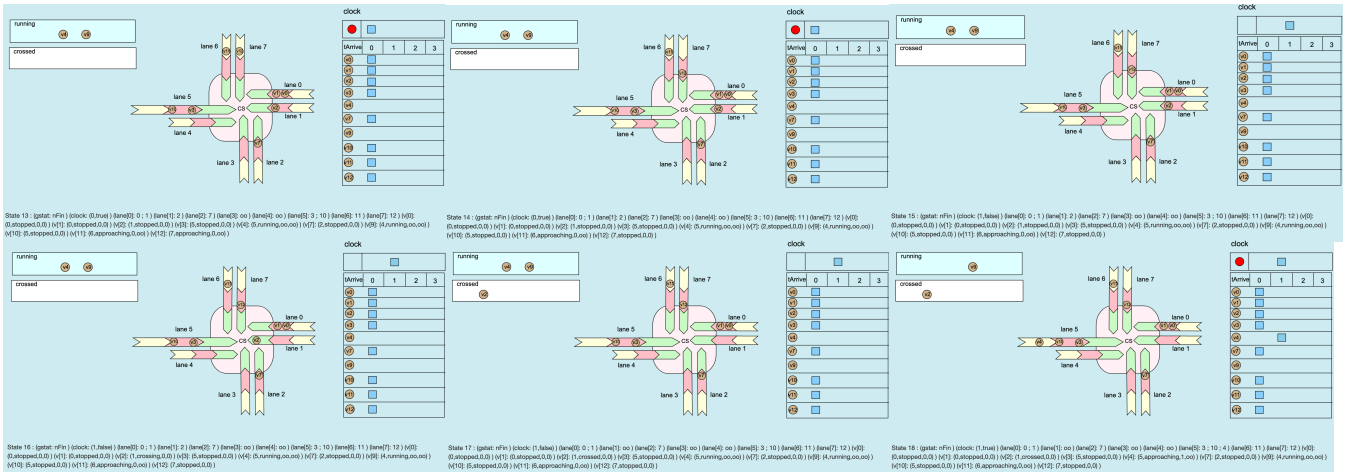


Figure 4. A state sequence for the LJPL protocol (1)

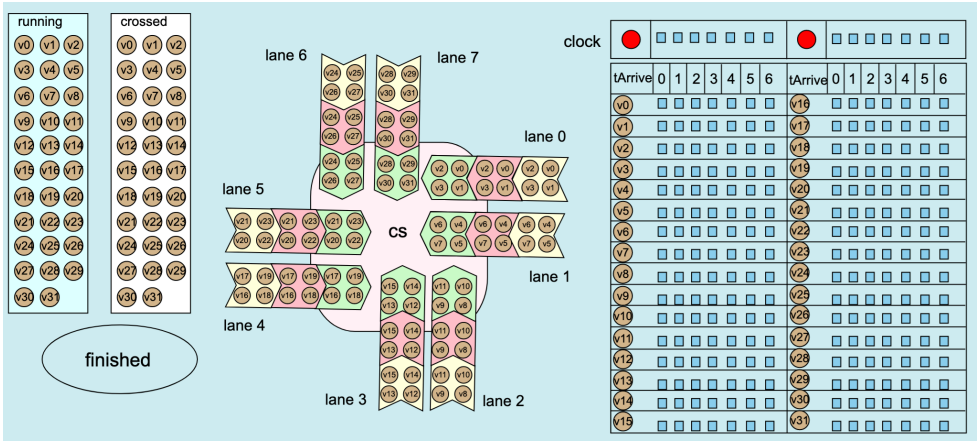


Figure 5. A flexible state picture design for the LJPL protocol (1)

vehicle cannot be greater than the first value of clock. The characteristic can be confirmed by Maude search command as follows:

```
search [1] in RIMUTEX :
init =>* {(clock: X:NatInf, false)
(v[i:NatInf]: j:NatInf, k:VStat,
Y:NatInf, t:NatInf) OCs}
such that
Y:NatInf >= X:NatInf and Y:NatInf =/= oo .
```

The search command above tries to find a reachable state in which the value of the time arrival of the vehicle `i` (i.e., `Y`) is greater than or equal to the first value of clock (i.e., `X`). Maude does not find any reachable state from the state `init` that satisfies the condition. Therefore, the guessed characteristic is confirmed with the initial state shown in Fig. 3.

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

```
search [1] in RIMUTEX :
init =>* {(clock: X:NatInf, b:Bool)
(v[i:NatInf]: j:NatInf, K:VStat,
Y:NatInf, t:NatInf) OCs}
such that
Y:NatInf == X:NatInf and K:VStat == running .
```

The search command above tries to find a reachable state in which the value of the time arrival of a vehicle `i` is equal to the first value of clock, and the status of the vehicle `i` is running. Maude does not find any reachable state from the state `init` that satisfies the condition. Consequently, the guessed characteristic is confirmed with the initial state shown in Fig. 3.

V. Lesson Learned

Through the case study with the LJPL protocol, we obtain several lessons on how to design a good state picture so that

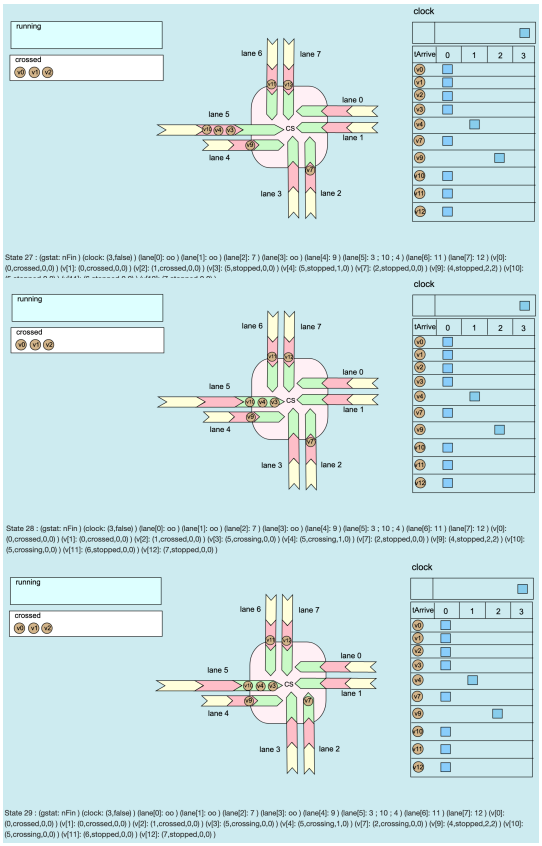


Figure 6. A state sequence for the LJPL protocol (2)

we can conjecture some non-trivial characteristics, especially when there are some observable components with composite values. Some of 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 some observable components that have the same values, we should design and display their domain values together in a 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.

VI. Conclusion

We have revised SMGA to be able to visualize composite data that are used in a state machine formalizing the LJPL protocol so that the LJPL protocol can be graphically animated. Some characteristics of the protocol have been conjectured by human users and confirmed with Maude based on our proposed state picture design. We have summarized our experiences as some tips on how to make a good state picture design for a state machine in which composite data are used. One future direction is to apply our work to other self-driving vehicle protocols, such as a merging protocol [12].

References

[1] T. T. T. Nguyen and K. Ogata, "Graphical animations of state machines," in *15th DASC*, 2017, pp. 604–611.

[2] K. W. Brodlie, et al., Ed., *Scientific Visualization: Techniques and Applications*. Springer, 1992.

[3] D. D. Bui and K. Ogata, "Better state pictures facilitating state machine characteristic conjecture," *MTAP*, 2021.

[4] M. Clavel, et al., Ed., *All About Maude*, ser. LNCS. Springer, 2007, vol. 4350.

[5] D. D. Bui and K. Ogata, "Graphical animations of the Suzuki-Kasami distributed mutual exclusion protocol," *JVLC*, vol. 2019, no. 2, pp. 105–115, 2019.

[6] M. Alpuente, et al., "Debugging Maude programs via runtime assertion checking and trace slicing," *J. Log. Algebraic Methods Program.*, vol. 85, no. 5, pp. 707–736, 2016.

[7] J. Lim, et al., "An efficient distributed mutual exclusion algorithm for intersection traffic control," *J. Supercomput.*, vol. 74, no. 3, pp. 1090–1107, 2018.

[8] M. N. Aung, Y. Phyo, and K. Ogata, "Formal specification and model checking of the Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol (S)," in *SEKE 2019*, 2019, pp. 159–208.

[9] D. D. Bui and K. Ogata, "Better state pictures facilitating state machine characteristic conjecture," in *DMSVIVA 2020*, 2020, pp. 7–12.

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

[11] D. Todorovic, "Gestalt principles," *Scholarpedia*, vol. 3, no. 12, p. 5345, 2008.

[12] S. Aoki and R. Rajkumar, "A merging protocol for self-driving vehicles," in *ICCPS 2017*, 2017, pp. 219–228.