Formal Verification of the Lim-Jeong-Park-Lee Autonomous Vehicle Control Protocol using the OTS/CafeOBJ Method

Tatsuya Igarashi Masaki Nakamura Kazutoshi Sakakibara

Toyama Prefectural University, Toyama, Japan

Abstract- **The Lim-Jeong-Park-Lee protocol (LJPL protocol) has been proposed as an efficient distributed mutual exclusion algorithm for intersection traffic control. The LJPL protocol has been specified and verified formally using the Maude model checker. Because of the limitation of computation, the existing model checking approach restricts the number of vehicles participating the protocol. In this paper, we model the LJPL protocol as an observational transition system, describe its specification in CafeOBJ, the algebraic specification language, and verify its safety property using the proof score method, where mutual exclusiveness can be proved for an arbitrary number of vehicles [*].**

Keywords-component; autonomous vehicles; the Lim-Jeong-Park-Lee protocol; algebraic specification; observational transition system; proof score method

I. INTRODUCTION

In [1], an efficient distributed mutual exclusion algorithm for intersection traffic control, called the Lim-Jeong-Park-Lee protocol (LJPL protocol), has been proposed, where each lane of the intersection has a queue of vehicles (Figure 1). All vehicles in a queue can enter the intersection if the top vehicle of the queue arrived first among the waiting vehicles in the other conflict lanes. Since the vehicles except the top one do not need extra permissions to enter the intersection, the protocol has been shown to be effective.

In [2], the LJPL protocol has been formally specified and some properties are verified by Maude tool[†]. In Maude, a state transition system is specified as a rewrite specification. A desired property is verified by fully automated model checking. In principle, model checking restricts the state space finite. Thus, by the Maude model, only finite combinations of initial states can be treated. In [2], it is mentioned that an initial state with five vehicles has been proved to be safe and the authors had encountered the state explosion problem for the case of more than a dozen vehicles.

In our study, we model the LJPL protocol as an observational transition system (OTS) [3, 4, 5], where a state of the system is not represented explicitly but can be identified through a given set of observation functions. A state transition is also defined through observations. By such an approach, we may obtain more abstract system specifications independent from the structure of states. Especially our model does not fix the number of vehicles participating the protocol. An OTS can be specified in CafeOBJ language[‡], which supports not only specification description based on equational specifications but also specification execution based on term rewriting theory. Roughly speaking, when we add a new equation $t_0 = t_1$ to a given specification SP, reduce a term t_2 by the CafeOBJ processor and obtain t_3 as a reduced term, then it guarantees that the implication $t_0 = t_1 \Rightarrow t_2 = t_3$ holds for all models of the specification. By combining specification executions, we may construct complicated proofs, such as case splitting and inductions. To make a complete proof through interaction with CafeOBJ processor is called the proof score method, or the OTS/CafeOBJ method. For the OTS/CafeOBJ specification of the LJPL protocol, we verify the safety property such that vehicles of different conflict lanes cannot enter at same time by the proof score method.

II. LIM-JEONG-PARK-LEE PROTOCOL

We give a brief introduction of the LJPL protocol in this section. See [1, 2] for more detail.

The intersection of the LJPK protocol is a crossroad represented in Figure 1. The lanes are labeled by lane0 , . . . , lane7. Each of four directions has two lane: the straight or right turn lanes (even numbered) and the left turn lanes (odd numbered). When some vehicle is crossing the intersection, some vehicle can enter the intersection and some are not. A lane l conflicts with a lane l' if a vehicle in l may collide with a vehicle in l' when they enter the intersection at same time. For example, lane0 conflicts with lane2, lane5, lane6 and lane7.

---

[*] DOI reference number: 10.18293/SEKE2022-028.

This work was supported by JSPS KAKENHI Number JP19K11842.

[†] http://maude.cs.uiuc.edu

[‡] https://cafeobj.org/intro/ja/

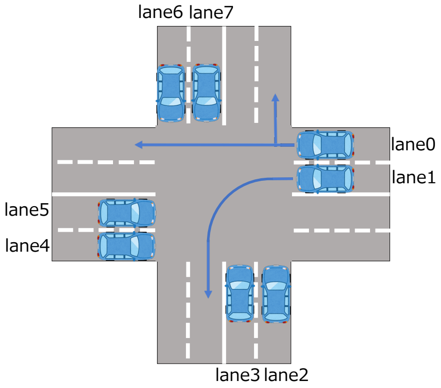

Figure 1. The intersection of the LJPL protocol

In the LJPL protocol, a vehicle passes the intersection through the following states: *running*, *approaching*, *stopped*, *crossing* and *crossed*. In the running state, the vehicle is running before the intersection. From the *running* state to the *approaching* state, the vehicle approaches the queue of a lane. From *approaching* to *stopped*, the vehicle is added to the queue and the arrival time of the top vehicle of the queue is set to the vehicle. From *stopped* to *crossing*, the vehicle enters the intersection if the time of the vehicle is less than the time of the top vehicle of each conflict lane. From *crossing* to *crossed*, the vehicle leaves the intersection.

III. An OTS/CafeOBJ specification of the LJPL protocol

In this section, we give an OTS/CafeOBJ specification of the LJPL protocol. We assume the reader is familiar with observational transition systems and CafeOBJ algebraic specification language, and introduce the notions and notations briefly through the specification of the LJPL protocol. See [3, 4, 5] for full syntax and semantics of OTS/CafeOBJ specifications.

A Data modules

An OTS/CafeOBJ specification consists of data modules and a system module. We first give a data module `VID` for vehicles.

```
mod* VID{
  [Vid < Vid+]
  op dummy : -> Vid+
  op _=_ : Vid+ Vid+ -> Bool {comm}
  eq (I:Vid = dummy) = false .
  eq (V:Vid+ = V) = true . }
```

The module declaration with `mod*` denotes the loose denotation, where the module denotes all models (algebras) which satisfies all equations in the modules. The name of the module is `VID`. Two sorts `Vid` and `Vid+` are declared with a relationship `Vid` < `Vid+`. Each sort denotes a (carrier) set in a model. Hereafter we deal with a sort as a set if no confusion occurs. Sort `Vid` is interpreted as a subset of `Vid+`. We intend to use `Vid` as a set of (identifiers of) vehicles and `Vid+` as a set of vehicles including a dummy one. There are two operator declarations with `op`. The name of the first operator is `dummy`, which takes the empty arity and returns `Vid+`. The empty arity operator denotes a constant of the returned sort. The commutative operator _=_ takes two arguments of `Vid+` and returns a boolean value. There are two equations which all models satisfy. A term X`:`S is a variable of Sort S, which denotes an arbitrary element of the sort. The first equation declare that all elements of `Vid` are not equivalent to the dummy vehicle. By the second equation, each vehicle is equivalent to itself.

The following is a data module for (identifiers of) lanes. For all $i, j \in \{0, 1, \ldots, 7, 999\}$, we give the values of `lane`i = `lane`j and `lane`i < `lane`j by equations. The dots part (`...`) are omitted.

```
mod! LID{
  [Lid]
  ops lane0 lane1 lane2 lane3 lane4 lane5 lane6 lane7
      lane999 : -> Lid
  op _=_ : Lid Lid -> Bool {comm}
  op _<_ : Lid Lid -> Bool
  eq (L:Lid = L) = true .
  eq (lane0 = lane1) = false .  ...
  eq (L:Lid < L) = false .
  eq (lane0 < lane1) = true . ...
  eq (lane5 < lane3) = false . ... }
```

The module declaration with `mod!` denotes the tight denotation, where the module denotes only the initial model, where each element of the model has a corresponding term constructed from operators in the module (no duplication), and an equation is deducible from the equations of the module, whenever the both hand sides of the equation are interpreted into a same element (no confusion). In the model of `LID`, Sort `Lid` has exactly nine elements of `lane0`∼`lane999`. The constants `lane0`∼`lane7` stand for lanes of the intersection. The constant `lane999` stands for a special lane where all vehicles belong before coming the intersection. We define the order of lanes as `lane` i is smaller than `lane` j iff $i < j$.

We specify a tight data module `VSTAT` of the labels of vehicles' states, where Sort `Vstat` and constants `running`, `approaching`, `stopped`, `crossing` and `crossed` of Sort `Vstat` are declared with an equivalent predicate _=_ similarly. We also specify a data module `TIMEVAL` with the built-in sort `Rat` of rational numbers, Sort `Rat+` of rational numbers with the infinity `oo` and predicates _<_, _<=_, _=_ on `Rat+`. We omit the details of `VSTAT` and `TIMEVAL`.

Since the LJPL protocol manages a queue of vehicles, we specify a data module `QUEUE` of queues as follows:

```
mod! QUEUE{
  pr(VID)  [Queue]
  op empty : -> Queue
```

```
  op _,_ : Vid Queue -> Queue
  op put : Queue Vid -> Queue
  op remove : Queue -> Queue
  op top : Queue -> Vid+ ... }
```

Module `QUEUE` imports Module `VID` with the protect mode, where a model (carrier sets) of the importing module includes a model of the imported module as it is. The first two operators `empty` and `_,_` are constructors of queues. The set `Queue` of queues is defined inductively. Constant `empty` denotes the empty queue. Term e`,`$queue$ is a queue whose top element is e and the tail is $queue$ if e is of `Vid` and $queue$ is of `Queue`. For example, Term e_0`,`e_1`,`e_2`,empty` is a term of `Queue`. Operator `put`, `remove` and `top` are standard operations of queues. Term `put(queue,e)` stands for the result queue by adding an element e to a $queue$ as the last element. Term `remove(queue)` is the result queue by deleting the top of $queue$. Term `top(queue)` is the top element of $queue$. For example, Operator `put` is defined by the following equations in `QUEUE`:

```
  eq put(empty,I:Vid) = I,empty .
  eq put((J:Vid,Q:Queue),I:Vid) = J,put(Q,I) .
```

For example, Term `put((a,b,empty),c)` is equivalent to Term `a,b,c,empty` since `put((a,b,empty),c) = a,put((b,empty),c) = a,b,put(empty,c) = a,b,c,empty`. Similarly, the other operators are defined inductively.

B The system module : observers

We give a system module of the LJPL protocol. First, we give observers and a definition of an initial state of our system module.

```
mod* OTS{
  pr(LID + VSTAT + QUEUE + TIMEVAL)
  *[Sys]*
  bop lid : Sys Vid+ -> Lid
  bop vstat : Sys Vid+ -> VStat
  bop t : Sys Vid+ -> Rat
  bop lt : Sys Vid+ -> Rat+
  bop q : Sys Lid -> Queue
  bop now : Sys -> Rat
```

Our system module, `OTS`, imports all data modules defined above with the protecting mode. Sort `Sys` is declared as a hidden sort, which denotes the state space of the system. An operator with the hidden sort in its arity is called a behavioral operator. A behavioral operator is divided into two categories: it is called an observer if the returned sort is not hidden and a transition if it is hidden. Six observers are declared in `OTS`: `lid(`s`,`i`)` is the lane ID of a vehicle i at a state s. `vstat(`s`,`i`)`, `t(`s`,`i`)` and `lt(`s`,`i`)` are the state, the arrival time, and the arrival time of the top vehicle in the queue of the lane of a vehicle i at a state s respectively. `q(`s`,`l`)` is the queue of a lane l. `now(`s`)` is the elapsed time at a state s.

The following specifies an initial state.

```
  op init : -> Sys
  eq lid(init,I:Vid) = lane999 .
  eq vstat(init,I:Vid) = running .
  eq t(init,I:Vid) = oo .
  eq lt(init,I:Vid) = oo .
  eq q(init,L:Lid) = empty .
  eq now(init) = 0 .
```

Constant `init` is an element of `Sys`, which we call the initial state. The initial state is not defined explicitly but is defined through observers. The first equation specifies the initial lane of all vehicles is `lane999`. The state is defined as `running`. The arrival times are defined as the infinity `oo`, which means that they have the lowest precedence to enter the intersection.

For the dummy vehicle, its lane, state, arrival times are defined as `lane999`, `stoped` and `oo` for all states respectively. The queue of `lane999` is defined as `empty` for all states. We omit the equations for the dummy vehicle.

C The system module : transitions

State transitions are declared as follows:

```
  bop set : Sys Vid Lid -> Sys
  bop approach : Sys Vid -> Sys
  bop check : Sys Vid -> Sys
  bop enter : Sys Vid -> Sys
  bop leave : Sys Vid -> Sys
  bop tick : Sys Rat -> Sys
```

Term `set(`s`,`i`,`l`)` is the result state after applying the transition `set` for a vehicle i and a lane l at the state s. Similarly, other transitions are declared as operators which take a current state and return the result state with some parameters. Transition `tick(`s`,`x`)` is a special transition which advances elapsed time by x.

For a transition τ, the effective condition `c-`τ is a condition under which the transition τ can be applied. The following is a definition of the effective condition `c-set` of the transition `set`[§].

```
  op c-set : Sys Vid -> Bool
  eq c-set(S:Sys,I) =
     (vstat(S,I) = running && lid(S,I) = lane999) .
  ceq set(S,I,L) = S if not c-set(S,I) .
```

The operator `c-set` is declared and is defined by the first equation such that `set` is effective for a vehicle `I` if `I`'s state is `running` and lane is `lane999`. The last equation is a conditional equation, where the body equation holds when the condition part is true. The condition part of the last equation is `not c-set(S,I)`, that is, `set` is not effective for `I`. Then, the body equation says that the result of applying `set` does not change a state. The application of the transition is considered to be ignored when it is not effective.

The following is the set of all equations defining `set` when it is effective.

[§] Hereafter we use `D`, `I`, `J`, `S` and `L` (`L'`) as variables of `Rat`, `Vid`, `Vid+`, `Sys` and `Lid` respectively.

```
  ceq lid(set(S,I,L),J) =
      (if I = J then L else lid(S,J) fi)
      if c-set(S,I) .
  ceq vstat(set(S,I,L),J) = vstat(S,J) if c-set(S,I) .
  ceq t(set(S,I,L),J) = t(S,J) if c-set(S,I) .
  ceq lt(set(S,I,L),J) = lt(S,J) if c-set(S,I) .
  ceq q(set(S,I,L),L') = q(S,L') if c-set(S,I) .
  ceq now(set(S,I,L)) = now(S) if c-set(S,I) .
```

The first equation specifies that the lane ID of a vehicle `J` after `set(S,I,L)` is defined as `L` if `I` = `J` when `set` is effective, and it is unchanged if `I` ≠ `J`. Only lane ID is changed and the other observed values are unchanged as defined by the following five equations. By Transition `set`, a vehicle can be assigned to any lane.

To define a system behavior completely, for all combinations of an observer o and a transition τ, we need to define the value observed by o of the result state after applying τ to a state s, denoted by $o(\tau(s))$. Since there are lots of equations in our system module, we show subset of them in this paper.

Transition `approach` is defined as follows:

```
eq c-approach(S,I) =
  (vstat(S,I) = running && not(lid(S,I) = lane999)) .
ceq vstat(approach(S,I),J) = (if I = J then approaching
    else vstat(S,J) fi) if c-approach(S,I) .
ceq t(approach(S,I),J) = (if I = J then now(S)
    else t(S,J) fi) if c-approach(S,I) .
ceq q(approach(S,I),L) = (if L = lid(S,I) then
    put(q(S,L),I) else q(S,L) fi) if c-approach(S,I) .
```

Transition `approach` is effective if the state is in `running` and the lane is not `lane999`, that is, immediately after `set`. By `approach(S,I)`, the state of `I` becomes `approaching`. The arrival time is set to the current time `now(S)` and `I` is added to the queue of the belonging lane.

Transition `check` is defined as follows:

```
eq c-check(S,I) = (vstat(S,I) = approaching &&
      top(q(S,lid(S,I))) = I
   || vstat(S,getpre(q(S,lid(S,I)),I)) = stopped
   || vstat(S,getpre(q(S,lid(S,I)),I)) = crossing)) .
ceq vstat(check(S,I),J) = (if I = J then stopped
    else vstat(S,J) fi) if c-check(S,I) .
ceq lt(check(S,I),J) = (if
    I = J && (top(q(S,lid(S,I))) = I
    || vstat(S,getpre(q(S,lid(S,I)),I)) = crossing)
    then t(S,I) else if
    I = J && vstat(S,getpre(q(S,lid(S,I)),I)) = stopped
    then lt(S,getpre(q(S,lid(S,I)),I)) else lt(S,J)
    fi fi) if c-check(S,I) .
```

Transition `check` is effective when the state of the vehicle is `approaching` and either it is top of the queue or the previous vehicle's state is `stopped` or `crossing`, where `getpre(`q`,`i`)` returns the previous vehicle of i in a queue q. The state of the vehicle becomes `stopped`. The last equation specifies that the arrival time of the previous vehicle in the queue is set to the vehicle as `lt`.

Transition `enter` is defined as follows:

```
eq c-enter(S,I) = (vstat(S,I) = stopped
                  && top(q(S,lid(S,I))) = I && ...)
ceq vstat(enter(S,I),J) = (if lid(S,I) = lid(S,J) &&
    vstat(S,J) = stopped then crossing else
    vstat(S,J) fi) if c-enter(S,I) .
```

Transition `enter` is effective when the vehicle's state is `stopped`, it is top of the queue and the arrival time `lt` is smaller than that of the top vehicle of each conflict lane. We omit a part of the right-hand side of the first equation of `c-enter`. The states of all vehicles in the same queue (`stopped`) become `crossing`, that is, they enter the intersection at once.

Transition `leave` is defined as follows:

```
eq c-leave(S,I) =
  (vstat(S,I) = crossing && top(q(S,lid(S,I))) = I) .
ceq vstat(leave(S,I),J) = (if I = J then crossed
    else vstat(S,J) fi) if c-leave(S,I) .
ceq q(leave(S,I),L) = (if L = lid(S,I) then
    remove(q(S,L)) else q(S,L) fi) if c-leave(S,I) .
```

Transition `leave` is effective when the vehicle's state is `crossing` and it is top of the queue. The vehicle's state becomes `crossed` and it is removed from the queue.

Finally, Transition `tick` is defined as follows:

```
eq now(tick(S,D)) = now(S) + D .
```

Transition `tick(S,D)` is always effective and it increase the current time by `D`.

D Specification execution

The CafeOBJ reduction command reduces a term to a term equivalent to the input term based on term rewriting theory. The following is an example of reduction.

```
open OTS . ...
eq s1 = set(set(set(set(
        init,a,lane0),b,lane3),b2,lane3),c,lane4) .
eq s2 = approach(approach(approach(s1,a),b),c) .
eq s3 = approach(tick(s2,1),b2) .
eq s4 = check(check(check(check(s3,a),b),c),b2).
eq s5 = enter(enter(enter(s4,c),b),a) .
red vstat(s5,a) . red vstat(s5,b) .
red vstat(s5,b2) . red vstat(s5,c) .
close .
```

State `s1` is equivalent to a term obtained by applying four `set` transitions with vehicles `a`, `b`, `b2`, `c` with lanes `lane0`, `lane3`, `lane3`, `lane5` respectively. We apply Transition `approach` to vehicles `a`, `b`, `c`, and advance time by one time unit and apply `approach` to `b2` (State `s3`). Then, we apply `check` to all vehicles and apply `enter` to `a`, `b`, `c`. State `s5` is the result state. Note that lane `lane0` does not conflict with lane `lane3` and `lane4` but lane `lane3` conflict with `lane4`.

By the last four reduction commands, we check states of all vehicles. CafeOBJ returns `crossing` for `a` in `lane0`, `crossing` for `b` and `b2` in `lane3`, and `stopped` for `c` in `lane4`. Vehicles `a` and `b`, `b2` are `crossing` since they do not conflict with each other. Vehicle `c` failed to enter (from `stopped` to `crossing`) since `b` in a conflict lane has already entered. Although `b2`'s arrival time is later than `c`'s arrival time, `b2` entered since it belongs to the same queue with `b`.

IV. Formal verification of the LJPL protocol

In this section we verify the safety property of the LJPL protocol, that is, no two vehicles enter the intersection if they belong to conflict lanes, by using the proof score method. First we formalize a safe state by operators and equations.

```
mod INV{ pr(OTS) ...
eq concur(L,L') = ((L = L') ||
  (L = lane0 && (L' = lane1 || L' = lane3 || L' = lane4))
  || ... .
eq inv1(S,I,J) = (not(I = J)
   && vstat(S,I) = crossing && vstat(S,J) = crossing)
   implies concur(lid(S,I),lid(S,J)) . }
```

The first equation specifies a predicate `concur` such that `concur(L,L')` is true if lanes `L` does not conflict with `L'`. Then, the invariant property `inv1` is defined by the last equation. The invariant property `inv1(S,I,J)` is true if vehicles `I` and `J` do not belong to conflict lanes whenever `I` and `J` are different and their states are `crossing`. The invariant property is a state predicate. If `inv1(`s, i, j`)` is true for all states s reachable from the initial state and vehicles i and j, the LJPL protocol is safe. In OTS/CafeOBJ specifications, reachable states are represented by terms like $\tau_n(\cdots(\tau_1(\tau_0(\texttt{init}))))$, which stands for the result state after applying transitions $\tau_0, \tau_1, \ldots, \tau_n$ to the initial state in this order. Since reachable terms are infinite, we prove this claim by induction on the structure of reachable states. The base step is proved for the initial state `init` and the induction step is proved for $s' = \tau(s)$ for each transition τ with the assumption of `inv1(`s, i, j`)` as the induction hypothesis.

Base step The following is a fragment of a proof score, called a proof passage, for the base step.

```
open INV .
ops i j : -> Vid .
red inv1(init,i,j) .
close .
```

Constants `i` and `j` are declared as arbitrary vehicles. The reduction command `red` takes a term and returns a term reduced by using declared equations. CafeOBJ processor returns `true` as the result of the above reduction, that guarantees that the base step is proved successfully.

Induction step The following is a module for proving induction steps.

```
mod ISTEP{ pr(INV) ...
  ops s s' : -> Sys
  eq istep1(I,J) = inv1(s,I,J) implies inv1(s',I,J) . }
```

Constants `s` and `s'` are declared as arbitrary states. For each induction step of a transition τ, we declare an equation $\mathtt{s}' = \tau(\mathtt{s})$. Thus, in induction steps, we prove the implication `inv1(s,`i, j`)` $\Rightarrow$ `inv1(s',`i, j`)` for each vehicles i and j. Predicate `istep1` is declared for proving the implication.

The following is a proof passage for Transition `set` in the case that the effective condition is false.

```
open ISTEP .
ops i j k : -> Vid . op l : -> Lid .
eq c-set(s,k) = false .
eq s' = set(s,k,l) .
red istep1(i,j) .
close .
```

The above reduction returns `true`. Thus, if `set` is not effective, the induction step for `set` is proved. The following is the case that it is effective.

```
open ISTEP .
ops i j k : -> Vid .
op l : -> Lid .
eq vstat(s,k) = running .
eq lid(s,k) = lane999 .
eq s' = set(s,k,l) .
red istep1(i,j) .
close .
```

Note that we declare two equations instead of `c-set(s,k) = true`. They are same meaning from the definition of `c-set` in the system module. Unfortunately, the above reduction does not return `true`. The result of the reduction is a term like `(if (k = i) then l else lid(s,i) fi) = (if (k = j) then ...) ...`. This result means that CafeOBJ cannot prove the input property to be true or false. In such a case, we revise the proof passage such that CafeOBJ can prove it. Such a procedure is called an interactive theorem proving.

In this case, the result term includes `k = i`. If it is true or false, CafeOBJ may proceed reduction more. Thus, we apply a case splitting about `k = i`. We make two copies of the above failed proof passage, add equations `k = i` and `(k = i) = false` for each copy. Since `k = i` $\vee$ `(k = i) = false` = *true*, if the both copies return true then the original proof passage is true. If results are not true or false, we apply case splitting until it is reduced into true or false.

Lemma discovery If a proof passage returns false, there are two possibilities, the invariant property is not true or the considered state is unreachable from the initial state.

Consider the following proof passage which returns false.

```
open ISTEP .
ops i j k : -> Vid .
eq vstat(s,k) = stopped . eq top(q(s,lid(s,k))) = k .
eq lid(s,k) = lane0 .     eq top(q(s,lane0)) = k .
eq vstat(s,top(q(s,lane2))) = stopped . ...
eq s' = enter(s,k) .
eq i = k . eq (j = k) = false . eq lid(s,j) = lane2 .
eq (lid(s,j)= lane0) = false . eq vstat(s,j) = crossing .
red istep1(i,j) .
close .
```

In this case, the vehicle `i = k` is the top of Lane `lane0` and waits for `enter`. Although the vehicle `j` is `crossing` in `lane2`, the top of `lane2` is `stopped`. In the LJPL protocol, a vehicle in a queue should not be the state of `crossing` if the top vehicle of its lane is in the state of `stopped`. Thus, this case of the proof passage is considered to be unreachable state from the initial state.

To solve this proof passage, we introduce a lemma extracted from the unreachable state. The following is a lemma we introduce.

```
eq pred1(S,empty) = true .
eq pred1(S,(I,Q)) = (if vstat(S,I) = crossing
                      then false else pred1(S,Q) fi) .
eq pred2(S,empty) = true .
eq pred2(S,(I,Q)) = (if vstat(S,I) = stopped
                     then pred1(S,Q) else pred2(S,Q) fi) .
eq inv2(S,I) = pred2(S,q(S,lid(S,I))) .
```

Predicate `inv2` is the lemma we introduce and Predicates `pred1` and `pred2` are auxiliary predicates for defining the lemma. Predicate `pred1(S,Q)` is true if the queue `Q` does not have `crossing` vehicles. Predicate `pred2(S,Q)` is true if no `crossing` vehicles exist after any `stopped` vehicle. The invariant `inv2(S,I)` is defined by `pred2` with State `S` and the queue of the lane of Vehicle `I`. We add the invariant to the proof passage as the premise of the target implication as follows:

```
open ISTEP . ...
eq (lid(s,j)= lane0) = false . eq vstat(s,j) = crossing .
red inv2(s,j) implies istep1(i,j) .
close .
```

Then, the reduction does not return false. We proceed case splitting and lemma discovery repeatedly and all proof passages (cases) for `inv1` become true after introducing more two lemmata `inv3` and `inv4`.

```
eq inv3(S,I) =
  (vstat(S,I) = approaching || vstat(S,I) = stopped ||
   vstat(S,I) = crossing) implies (I in q(S,lid(S,I))) .
eq inv4(S,I) = vstat(S,I) = crossing implies
   not pred1(S,q(S,lid(S,I))) .
```

Verification of lemmata In the previous section we showed the main invariant property `inv1` holds under the assumption of three lemmata. In order to complete a proof we need to prove those lemmata. They can be proved by the induction on reachable states similarly. Although we do not need more lemmata about the induction on reachable states, we needed to introduce another kind of lemmata, for example, `pred2(set(s,k,L),q)` = `pred2(s,q)`, which can be proved by the induction on the data structure of queues `q`.

Finally, we obtain a complete proof score for `inv1` with 609 proof passages which all return true, where three lemmata about reachable states and 17 lemmata about queues are introduced. Since the data module `VID` of vehicles denotes the loose denotation, the system specification `OTS` denotes all systems following the LJPL protocol with arbitrary number of vehicles. Our verification result guarantees that the LJPL protocol is safe for any vehicles.

V. CONCLUSION

We described an OTS model of the LJPL protocol in CafeOBJ language and verified a safety property by the proof score method. The main contribution of our study is to give a formal verification of the safety property of the LJPL protocol for arbitrary number of vehicles.

Through the experience of formal verification of the LJPL protocol, we faced lemmata about queues as well as lemmata `inv1`~`inv4` about reachable states. Although to find an appropriate lemma about reachable states we may need an insight into a target system, the lemmata about queues seem to have some pattern. To investigate a way to construct a semi-automated support tool for the proof score method for such data types is one of our future work.

In [2], not only the safety property we deal with in this study but other important properties of intersection control protocols have also been verified, e.g. the deadlock-freedom and the starvation-freedom properties. To specify and verify such properties in our OTS/CafeOBJ specification is another one of our future work.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP19K11842.

REFERENCES

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

[2] Moe Nandi Aung, Yati Phyo, Kazuhiro Ogata, Formal Specification and Model Checking of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol, SEKE 2019, pp.159-208, 2019.

[3] K. Ogata, and K. Futatsugi, Proof scores in the OTS/CafeOBJ method, FMOODS 2003, LNCS 2884, pp.170-184. Springer, 2003.

[4] K. Ogata and K. Futatsugi, Modeling and verification of real-time systems based on equations, Science of computer programming, 66(2), pp.162-180, 2007.

[5] Masaki Nakamura, Shuki Higashi, Kazutoshi Sakakibara, Kazuhiro Ogata, Specification and verification of multitask real-time systems using the OTS/CafeOBJ method, IEICE Transactions on Information and Systems, Vol.E105-A, No.5, pp.-, 2022. (accepted)