Graphical Animations of the Suzuki-Kasami Distributed Mutual Exclusion Protocol

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

***Abstract*—A state machine that formalizes a distributed mutual exclusion protocol called the Suzuki-Kasami protocol is graphically animated. The network with which messages are exchanged among nodes in the protocol is displayed as follows: some limited number of messages in the network are displayed and the others are depressed when there are many messages in the network. We prepare one place dedicated to the message that has been just put into the network by a node (or just sent by a node) and one place dedicated to the message that has been just received by a node (just deleted from the network by a node). The main purpose of graphically animating state machines is to make it possible for humans to perceive characteristics or properties of the state machines by observing their graphical animations. We can guess some properties of the state machine formalizing the Suzuki-Kasami protocol by observing graphical animations of the state machine and confirm them by model checking, which demonstrates that state machine graphical animations could make humans perceive state machine properties.**

***Keywords*-distributed protocols; graphical animations; Maude; model checking; SMGA; state machines;**

I. Introduction

Many kinds of information and communication technology (ICT) systems can be formalized as state machines. Some ICT systems, such as the Internet, have become important societal infrastructures, they must be highly reliable. It is crucial to comprehend ICT systems better so as to make them highly reliable. Because ICT systems can be formalized as state machines, one possible way to comprehend ICT systems is to understand state machines that formalize the ICT systems. There may be multiple possible ways to understand state machines, but one promising way to do so would be to rely on human visual perception. Therefore, Nguyen and Ogata [1] have developed a tool called SMGA that generates graphical animations of state machines. Some shared-memory mutual exclusion protocols and some communication protocols have been tackled with SMGA so far. But, ICT systems that are societal infrastructures are often in the form of distributed systems. In this paper, thus, a distributed mutual exclusion protocol called the Suzuki-Kasami protocol [2] is tackled with SMGA.

As all other distributed systems, the Suzuki-Kasami protocol uses a network with which messages are exchanged among nodes. The network is expressed as an associative-commutative collection (called a soup) of messages in a state machine formalizing the Suzuki-Kasami protocol. Because there may be a lot of messages in the network, it may be impossible to display all of the messages on a limited space. Our idea is to display a limited number of messages and suppress the others. The message that has been just put into the network (just sent by a node) or just deleted from the network (just received by a node) is crucial information and then we prepare one place dedicated to the one that has been just put into the network and one place dedicated to the one that has been just deleted from the network.

Understanding a state machine is to know properties the state machine enjoys. The more state machine properties we know, the better we understand the state machine. The main purpose of SMGA is to make it possible for humans to perceive characteristics or properties of a state machine by observing graphical animations of the state machine. Observing graphical animations of state machines, we could guess some properties of the state machines [3]. We guess some properties of the state machine formalizing the Suzuki-Kasami protocol by observing its graphical animations and confirm the properties by model checking. We use Maude [4], a rewriting logic-based computer language, as a specification language for state machines and a model checker.

The rest of the paper is organized as follows: § II Preliminaries, § III State Machine Graphical Animation (SMGA), § IV Suzuki-Kasami Protocol, § V Specification of Suzuki-Kasami Protocol, § VI Graphical Animations of Suzuki-Kasami Protocol, § VII Confirmation of Guessed Properties with Model Checking, § VIII Related Work and § IX Conclusion.

II. Preliminaries

A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states, the 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 $s \to_M s'$. The set $R_M \subseteq S$ of reachable states w.r.t. M is inductively defined as follows: (1) for each $s \in I$, $s \in R$ and (2) if $s \in R$ and $(s, s') \in T$,

This work was partially supported by JSPS KAKENHI Grant Number JP26240008 & JP19H04082.

DOI reference number: 10.18293/DMSVIVA2019-012

then $s' \in R$. A state predicate p is called invariant w.r.t. M if and only if $p(s)$ holds for all $s \in R_M$. 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 $i = 0, \ldots, n-1$.

There are multiple possible ways to express states. We express a state as a braced associative-commutative (AC) collection of name-value pairs. AC collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a braced soup of observable components. The juxtaposition operator is used as the constructor of soups. Let oc_1, oc_2, oc_3 be observable components, and then $oc_1\ oc_2\ oc_3$ is the soup of those three observable components. A state is expressed as $\{oc_1\ oc_2\ oc_3\}$. There are multiple possible ways to specify state transitions. We specify them as rewrite rules. Concretely, we use Maude [4], a programming/specification language based on rewriting logic. Maude makes it possible to specify complex systems flexibly and is also equipped with model checking facilities (a reachability analyzer and an LTL model checker).

Let us consider as an example a test&set mutual exclusion (or spin-lock) protocol whose pseudo-code is as follows:

```
Loop:
    "Remainder Section"
  rm : repeat while test&set($locked$);
    "Critical Section"
  cs :$locked$ := false;
```

$locked$ is a Boolean variable shared with all processes participating in the protocol. test&set($locked$) atomically performs the following: it sets $locked$ to true and returns the old value stored in $locked$. Each process is located at either rm (Remainder Section) or cs (Critical Section). Initially, each process is located at rm and $locked$ is false. When a process wants to enter Critical Section, it repeatedly conducts test&set($locked$) until false is returned and then goes to cs. When it leaves the critical section, it sets $locked$ to false and goes back to rs.

The protocol is formalized as a state machine M_{TS}. When there are three processes participating in the protocol, a state in S_{TS} is expressed as follows:

```
{(pc[p1]: l1)(pc[p2]: l2)(pc[p3]: l3)
 (locked: b)}
```

where li (for $i = 1, 2, 3$) is either rs or cs and b is either true or false. I_{TS} consists of one state that is expressed as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (locked: false)}
```

which will be referred as `ic`. T_{TS} is specified by the following two rewrite rules:

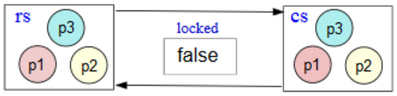


Figure 1. A picture of states in S_{TS}

```
rl [enter] : {(locked: false) (pc[I]: rs) OCs}
=> {(locked: true) (pc[I]: cs) OCs} .

rl [exit] : {(locked: B) (pc[I]: cs) OCs}
=> {(locked: false) (pc[I]: rs) OCs} .
```

Rewrite rules are defined with `rl`, while conditional ones are defined with `crl` and their conditions are written after `if`. `enter` and `exit` are labels (or names) given to the rules, respectively. `I` is a Maude variable of process identifications, `B` is a Maude variable of Boolean values and `OCs` is a variable of observable component soups.

III. State Machine Graphical Animation (SMGA)

SMGA [1] basically takes a finite computation of a state machine and generates a graphical animation of the state machine. For each state, a picture is designed and then we get a series of pictures from a finite computation. Such a series of pictures is regarded as a movie film. This is how SMGA generates a graphical animation of a state machine.

We could design a picture of states in S_{TS} as shown in Fig. 1 when there are three processes. Fig. 2 shows a sequence of pictures for M_{TS} generated by SMGA. Observing such a sequence of pictures or a graphical animation of a state machine, we could guess some properties of the state machine [3]. Observing the sequence of pictures shown in Fig. 2, for example, we could guess the property that $locked$ is false if and only if there is no process in the critical section, or equivalently $locked$ is true if and only if there exists a process in the critical section. The guessed property is invariant with respect to M_{TS}. Such a guessed property could be confirmed by model checking with Maude. We use the Maude reachability analyzer (or the search command). The following command can be used to confirm the guessed property of M_{TS}:

```
search [1] in TS : ic
=>* {(locked: false) (pc[I]: cs) OCs} .
```

Maude exhaustively searches the reachable states $R_{M_{\mathrm{TS}}}$ from the initial state `ic` for a state that can match `{(locked: false) (pc[I]: cs) OCs}` but does not find any such states. This means that the guessed property is invariant with respect to M_{TS} when there are three processes.

IV. Suzuki-Kasami Protocol

The Suzuki-Kasami protocol is a distributed mutual exclusion protocol [2]. The basic idea is that a node that has a privilege is only allowed to enter its critical section,

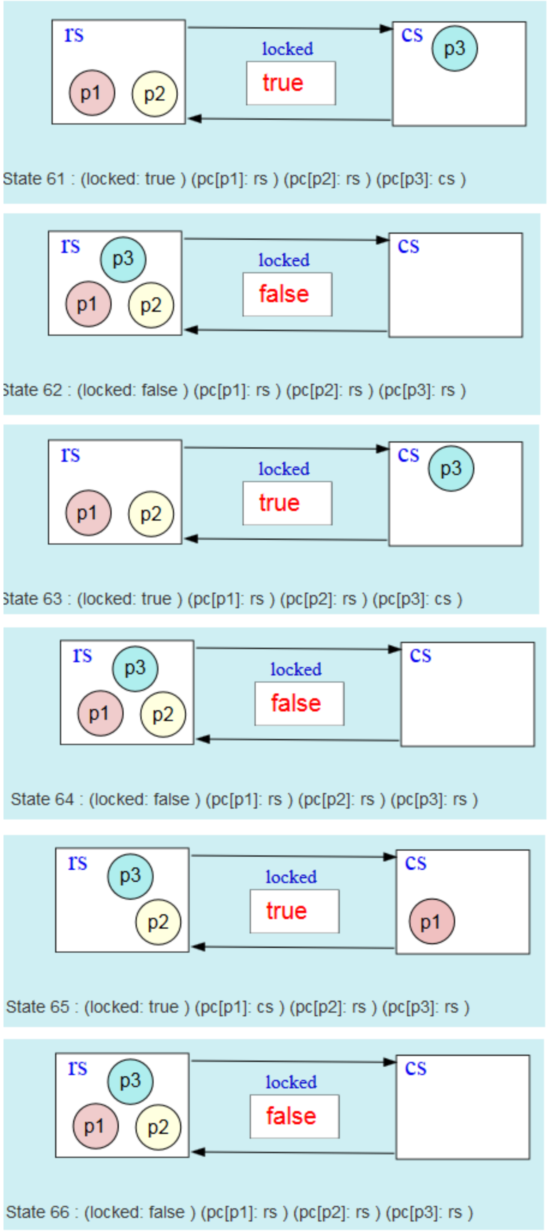

rs
p1
p2
locked
true
cs
p3
State 61 : (locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs)
rs
p3
p1
p2
locked
false
cs
State 62 : (locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
rs
p1
p2
locked
true
cs
p3
State 63 : (locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs)
rs
p3
p1
p2
locked
false
cs
State 64 : (locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
rs
p3
p2
locked
true
cs
p1
State 65 : (locked: true) (pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs)
rs
p3
p1
p2
locked
false
cs
State 66 : (locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)

Figure 2. A sequence of pictures for M_{TS}

and there exists one and only one privilege in the system. The privilege is owned by a node, or is in the network being transferred by a node to another. We suppose that N nodes participate in the protocol and the natural numbers $1, \ldots, N$ are used as their identifications. Let Node be $\{1, \ldots, N\}$. The N nodes have no memory in common and can communicate only by exchanging messages. The communication delay is totally unpredictable, namely that although messages eventually arrive at their destinations, they are not guaranteed to be delivered in the same order in which they are sent. There are two kinds of messages used in the Suzuki-Kasami protocol: request and privilege messages. A request message is in the from $\mathrm{request}(j, n)$, where j is the identification of the node that has sent the message and n is a request number. A privilege message is in the form $\mathrm{privilege}(q, a)$, where q is a queue of node identifications and a is a natural number array of size N.

The Suzuki-Kasami protocol consists of two procedures P1 and P2 for each node $i \in \mathrm{Node}$. The procedures for node i are shown in Fig. 3.

$request$ and $have\_privilege$ are Boolean variables. $request$ indicates whether or not node i wants to enter its critical section, and $have\_privilege$ indicates whether or not node i owns the privilege. $queue$ is a queue of Node. It contains the identifications of nodes that wait to enter their critical sections. ln and rn are natural number arrays of size N. $ln[j]$ for each node $j \in \mathrm{Node}$ is the sequence number of node j's request granted most recently. rn records the largest request number ever received from each of the other nodes. Node i uses $rn[i]$ to generate the sequence numbers of its own requests. For each node $i \in \mathrm{Node}$, its rn is always meaningful, while its $queue$ and ln are meaningful only when node i owns the privilege. When the privilege is in the network, $queue$ and ln contained in the privilege message are meaningful. For each node $i \in \mathrm{Node}$, initially, $request$ is false, $have\_privilege$ is true if $i = 1$ and false otherwise, $queue$ is empty, and $ln[j]$ and $rn[j]$ for each $j \in \mathrm{Node}$ are 0.

If node i wants to enter its critical section, it first calls its own procedure P1, which sets $request$ to true. If it happens to own the privilege, it immediately enters the critical section. Otherwise, it generates the next sequence number, namely, incrementing $rn[i]$, and sends the request message $request(i, rn[i])$ to all other nodes. When it receives a privilege message $privilege(queue, ln)$, it enters the critical section. When it leaves the critical section, it sets $ln[i]$ to its current sequence number $rn[i]$, meaning that the current request has been granted, and updates $queue$ such that if there are nodes that want to enter their critical sections and whose identifications are not yet in the queue, their identifications are added to the queue. After that, if $queue$ is not empty, node i sets $have\_privilege$ to false and sends the privilege message $privilege(\mathrm{deq}(queue), ln)$ to the node found in the front of the queue. Otherwise, node i keeps the privilege. Finally, node i sets $request$ to false and leaves procedure P1.

Whenever $request(j, n)$ is delivered to node i, node i executes its own procedure P2. However, procedure P2 has to be atomically executed. When node i executes procedure P2, it sets $rn[j]$ to n if n is greater than $rn[j]$. Then, if node i owns the privilege, does not want to enter its critical section, and the nth request of node j has not been granted, that is, $rn[j] = ln[j] + 1$, then it sets $have\_privilege$ to false and sends the privilege message $privilege(queue, ln)$ to node j.

```
try(i)      ⟷  rem   procedure P1
setReq(i)   ⟷  l1      requesting := true;
chkPrv(i)   ⟷  l2      if ¬have_privilege then
incRN(i)    ⟷  l3        rn[i] := rn[i] + 1;
                         for all j ∈ {1, …, N} − {i} do
sndReq(i)   ⟷  l4          send request(i, rn[i]) to node j;
                         endfor
                         wait until privilege(queue, ln) is received;
wtPrv(i)    ⟷  l5        have_privilege := true;
                       endif
exit(i)     ⟷  cs      Critical Section;
cmpReq(i)   ⟷  l6      ln[i] := rn[i];
                       for all j ∈ {1, …, N} − {i} do
                         if (j ∉ queue) ∧ (rn[j] = ln[j] + 1) then
updQ(i)     ⟷  l7          queue := enq(queue, j);
                         endif
                       endfor
chkQ(i)     ⟷  l8      if queue ≠ empty then
                         have_privilege := false;
trsPrv(i)   ⟷  l9        send privilege(deq(queue), ln) to node top(queue);
                       endif
                       request := false;
rstReq(i)   ⟷  l10   endproc

                     // request(j, n) is received; P2 is indivisible.
                     procedure P2
                       rn[j] := max(rn[j], n);
                       if have_privilege ∧ ¬request ∧ (rn[j] = ln[j] + 1) then
recReq(i)   ⟷            have_privilege := false;
                         send privilege(queue, ln) to node j;
                       endif
                     endproc
```

Figure 3. Suzuki-Kasami distributed mutual exclusion protocol in an Algol-like language

V. Specification of Suzuki-Kasami Protocol

Let Nat, Bool, Loc, NodeQueue, and NatNArray be the set of all natural numbers, the set of the Boolean values (true and false), the set of all locations, such as rem and l1, the set of all queues of Node, and the set of all natural number arrays whose sizes are N, respectively. A request message addressed to node $i \in \mathrm{Node}$ by node $j \in \mathrm{Node}$ is expressed as $\mathrm{msg}(i, \mathrm{req}(j, k))$, where $k \in \mathrm{Nat}$, msg is used as the constructor of messages and req is used as the constructor of requests. A privilege message addressed to node $i \in \mathrm{Node}$ is expressed as $\mathrm{msg}(i, \mathrm{priv}(q, a))$, where $q \in \mathrm{NodeQueue}$, $a \in \mathrm{NatNArray}$ and priv is used as the constructor of privileges. Let Req and Priv be $\{\mathrm{req}(i, n) \mid i \in \mathrm{Node}, n \in \mathrm{Nat}\}$ and $\{\mathrm{priv}(q, a) \mid q \in \mathrm{NodeQueue}, a \in \mathrm{NatNArray}\}$, respectively. The network is formalized as a soup of messages that are request and privilege messages. The set MsgSoup of all soups of messages is inductively defined as follows: void $\in$ MsgSoup, for each $i \in \mathrm{Node}$, $r \in \mathrm{Req}$, and $p \in \mathrm{Priv}$, $\mathrm{msg}(i, r) \in \mathrm{MsgSoup}$ and $\mathrm{msg}(i, p) \in \mathrm{MsgSoup}$, and for each $ms_1, ms_2 \in \mathrm{MsgSoup}$, $ms_1 \,;\, ms_2 \in \mathrm{MsgSoup}$. A semicolon **;** is used as the constructor of soups of messages. void denotes the empty soup of messages and is the identity of **;**, namely that $ms \,;\, \mathrm{void} = \mathrm{void} \,;\, ms = ms$ for each $ms \in \mathrm{MsgSoup}$. Each message is also treated as the singleton soup that only consists of the message.

The Suzuki-Kasami protocol is formalized as a state machine M_{SK}, which is specified in Maude. P1 is divided into 12 regions shown in Fig. 3. The 12 regions are referred as the 12 locations, such as rem and l1. We suppose that each node is at one of those 12 locations. Procedure P2 is regarded as one region and then there are totally 13 regions in the Suzuki-Kasami protocol. The 13 regions are given names, such as $\mathrm{try}(i)$ and $\mathrm{setReq}(i)$, shown at the left-most column in Fig. 3. For each node i, there are 13 kinds of transitions that corresponds to the 13 regions. The 13 region names are used to refer to the 13 kinds of transitions. Let

```
rl [try] : {(#req[I]: X) (pc[I]: rem) (tran: T) OCs}
=> {(#req[I]: (if X < M then X + 1 else X fi)) (pc[I]: (if X < M then l1 else rem fi))
    (tran: try(I)) OCs} .

rl [setReq] : {(pc[I]: l1) (request[I]: F) (tran: T) OCs}
=> {(pc[I]: l2) (request[I]: true) (tran: setReq(I)) OCs} .

rl [chkPrv] : {(pc[I]: l2) (havePriv[I]: F) (tran: T) OCs}
=> {(pc[I]: (if F then cs else l3 fi)) (havePriv[I]: F) (tran: chkPrv(I)) OCs} .

rl [incRN] : {(pc[I]: l3) (rn[I]: RN) (idx[I]: K) (tran: T) OCs}
=> {(pc[I]: l4) (rn[I]: RN[I] := (RN[I]) + 1) (idx[I]: 1) (tran: incRN(I)) OCs} .

rl [sndReq] : {(pc[I]: l4) (idx[I]: K) (rn[I]: RN) (nw: NW) (tran: T) OCs}
=> {(pc[I]: if K == N then l5 else l4 fi) (idx[I]: if K == N then 1 else K + 1 fi) (rn[I]: RN)
    (nw: (if K == I then NW else msg(K,req(I,RN[I])) ; NW fi)) (tran: sndReq(I)) OCs} .

rl [wtPrv] : {(pc[I]: l5) (havePriv[I]: F) (ln[I]: LN’) (queue[I]: Q’)
    (nw: (msg(I,priv(Q,LN)) ; NW)) (tran: T) OCs}
=> {(pc[I]: cs) (havePriv[I]: true) (ln[I]: LN) (queue[I]: Q) (nw: NW)
    (tran: wtPrv(I)) OCs} .

rl [exit] : {(pc[I]: cs) (tran: T) OCs} => {(pc[I]: l6) (tran: exit(I)) OCs} .

rl [cmpReq] : {(pc[I]: l6) (rn[I]: RN) (ln[I]: LN) (idx[I]: K) (tran: T) OCs}
=> {(pc[I]: l7) (rn[I]: RN) (ln[I]: LN[I] := RN[I]) (idx[I]: 1) (tran: cmpReq(I)) OCs} .

rl [updQ] : {(pc[I]: l7) (idx[I]: K) (rn[I]: RN) (ln[I]: LN) (queue[I]: Q) (tran: T) OCs}
=> {(pc[I]: if K == N then l8 else l7 fi) (idx[I]: if K == N then 1 else K + 1 fi)
    (rn[I]: RN) (ln[I]: LN)
    (queue[I]: if K =/= I and not(K \in Q) and (RN[K] == (LN[K]) + 1)
        then put(Q,K) else Q fi) (tran: updQ(I))} .

rl [chkQ] : {(pc[I]: l8) (queue[I]: Q) (tran: T) OCs}
=> {(pc[I]: if empty?(Q) then l10 else l9 fi) (queue[I]: Q) (tran: chkQ(I)) OCs} .

rl [trsPrv] : {(pc[I]: l9) (havePriv[I]: F) (ln[I]: LN) (queue[I]: Q) (nw: NW)
    (tran: T) OCs}
=> {(pc[I]: l10) (havePriv[I]: false) (ln[I]: LN) (queue[I]: Q)
    (nw: (msg(top(Q),priv(get(Q),LN)) ; NW)) (tran: trsPrv(I)) OCs} .

rl [rstReq] : {(pc[I]: l10) (request[I]: F) (tran: T) OCs}
=> {(pc[I]: rem) (request[I]: false) (tran: rstReq(I)) OCs} .

crl [recReq] : {(pc[I]: L) (request[I]: F) (havePriv[I]: F’) (rn[I]: RN) (ln[I]: LN)
    (queue[I]: Q) (nw: (msg(I,req(J,X)) ; NW)) (tran: T) OCs}
=> {(pc[I]: L) (request[I]: F) (havePriv[I]: if C then false else F’ fi)
    (rn[I]: RN[J] := Max) (ln[I]: LN) (queue[I]: Q)
    (nw: if C then (msg(J,priv(Q,LN)) ; NW) else NW fi) (tran: recReq(I)) OCs}
if I =/= J /\ L =/= l10 /\ L =/= l8 /\ L =/= l7 /\
  Max := if (RN[J]) < X then X else RN[J] fi /\
  C := F’ and not(F) and Max == (LN[J]) + 1 .
```

Figure 4. Rewrite rules specifying T_{SK}

us note that $\mathrm{sndReq}(i)$ and $\mathrm{updQ}(i)$ correspond to each iteration of the loops at labels l4 and l7, respectively.

When there are three nodes, a state in M_{SK} is expressed as follows:

{n(1) n(2) n(3) (nw: ms) (tran: t)}

where n(i) is as follows:

(#req[i]: n) (pc[i]: l) (request[i]: $b1$)
(havePriv[i]: $b2$) (rn[i]: $a1$) (ln[i]: $a2$)
(queue[i]: q) (idx[i]: j)

and where ms is a soup of messages in the network, t is the transition that has been just taken, n is the number of requests made by node i, l is the location where node i is, $b1$ is the value of the node i's $request$, $b2$ is the value of the node i's $have\_privilege$, q is the value of the node i's $queue$ and j is the value of the node i's j, a loop variable. The state expression defines S_{SK}.

When there are three nodes, I_{SK} consists of one state that is expressed as follows:

{n(1) n(2) n(3) (nw: void) (tran: notran)}

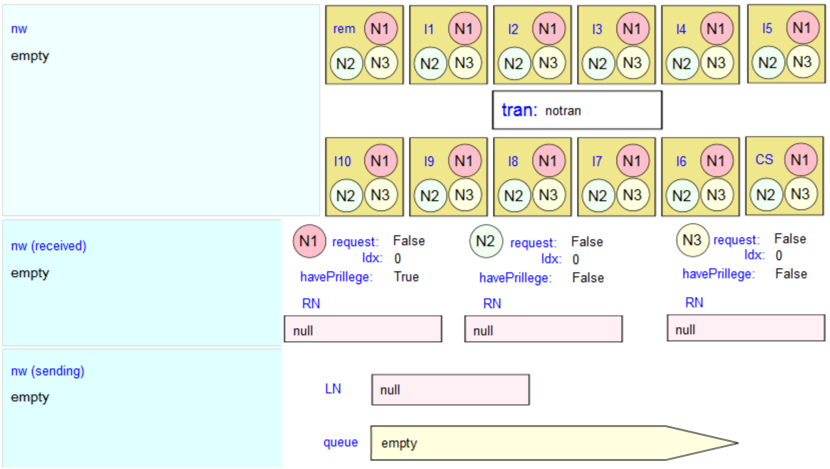


Figure 5. A picture of states in S_{SK}

where `notran` means that no transition has been taken and `n(I)` is as follows:

```
(#req[I]: 0) (pc[I]: rem) (request[I]: false)
(havePriv[I]: (I == 1)) (rn[I]: ia)
(ln[I]: ia) (queue[I]: empty) (idx[I]: 1)
```

where `ia` denotes the natural number array such that each slot is 0, `I == 1` is `true` if `I` is 1 and `false` otherwise, and `empty` denotes the empty queue.

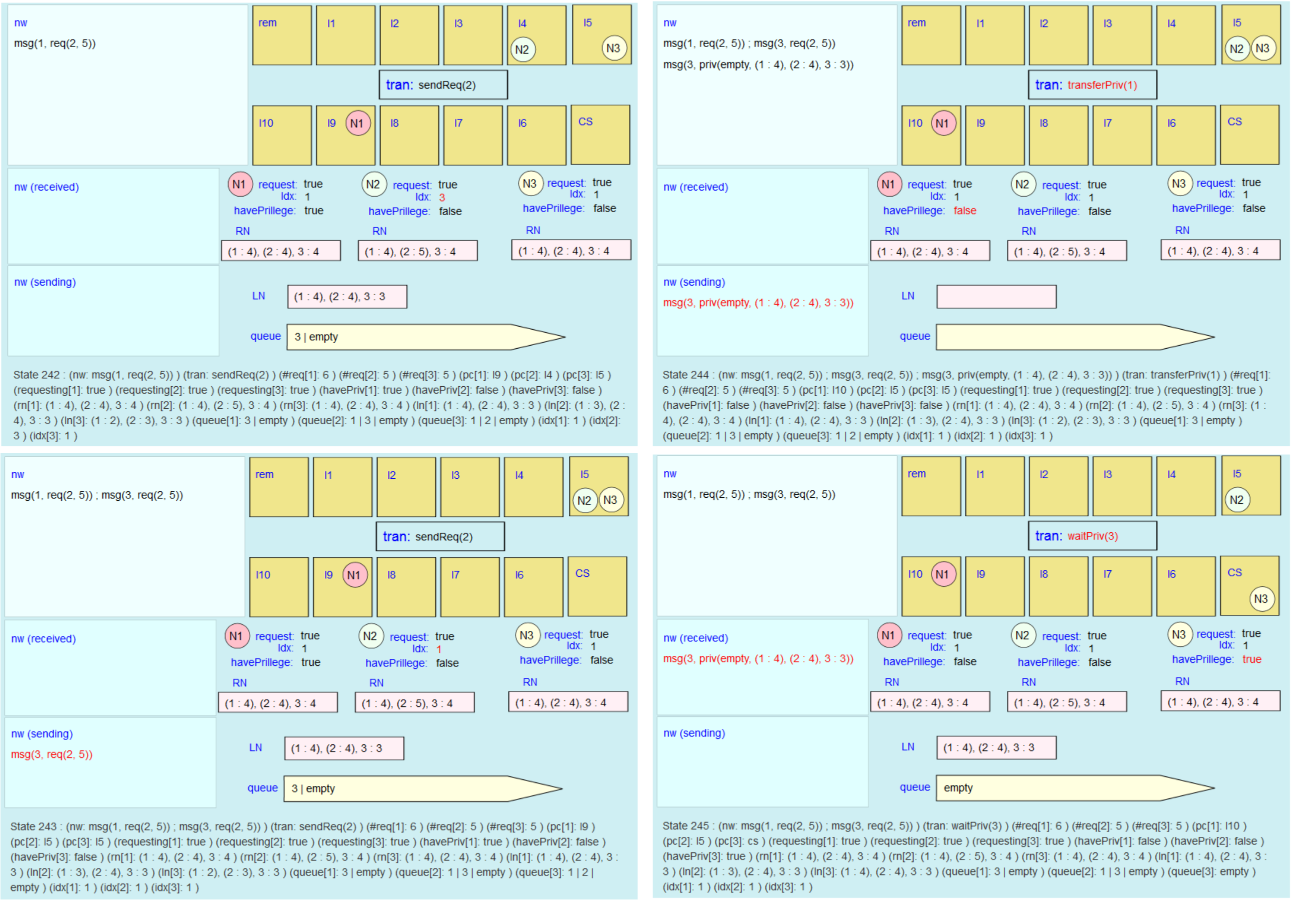
T_{SK} is specified in terms of (conditional) rewrite rules that are shown in Fig. 4. The words starting with a capital letter, such as `X`, `I`, `T` and `OCs`, are Maude variables. Their types (or sorts) could be understood from the context. For example, `X`, `I`, `T` and `OCs` are variables of Nat, Loc, transition names and observable component soups, respectively. `RN[I] := (RN[I]) + 1` is the array assignment at index `I`. `K \in Q` is the membership predicate `K` ∈ `Q` over queues. `put(Q,K)` denotes the queue obtained by putting `K` into `Q` at bottom. `top(Q)` is the top element of `Q`. `get(Q)` denotes the queue obtained by deleting the top from `Q`. What is called a matching equation[1] V `:=` T, where V is a fresh variable and T is a term, can be used in rule conditions and is like **let** expressions in functional programming languages.

This paper describes how to interpret four rewrite rules `sndReq`, `wtPrv`, `trsPrv` and `recReq`. The other rules can be interpreted likewise. Rewrite rule `sndReq` says that when node `I` is located at `l4`, the message `msg(K,req(I,RN[I]))` is put into the network, where the message is addressed to node `K`, unless `K` equals `I` because `I` does not need to send itself the message. Moreover, if `K` equals `N`, node `I` moves to `l5` and otherwise it stays `l4`. Rewrite rule `wtPrv` says that when node `I` is located at `l5` and there is a privilege message `msg(I,priv(Q,LN))` addressed to `I`, node `I` receives it, enters the critical section and sets its $have\_privilege$ to true. Rewrite rule `trsPrv` says that when node `I` is located at `l9`, it puts a privilege message `msg(top(Q),priv(get(Q),LN))` in the network, where the message is addressed to the node `top(Q)`, and sets its $have\_privilege$ to false. Rewrite rule `recReq` says that when there is a request message `msg(I,req(J,X))` addressed to node `I` in the network and the designated condition is fulfilled, then node `I` puts a privilege message `msg(J,priv(Q,LN))` in the network and sets its its $have\_privilege$ to false. Among the designated condition is that node `I` is not located at `l7`, `l8` or `l10`. This is because otherwise a livelock may occur [5], [6], [7].

VI. Graphical Animations of Suzuki-Kasami Protocol

Fig. 5 shows a picture of states in S_{SK} when there are three nodes participating in the Suzuki-Kasami protocol. There is a pane (called the nw pane) located in the left upper corner where the messages in the network are displayed. Under the nw pane, there is a pane (called the nw (received) pane) where the message that has been just received by a node (or just deleted from the network) is displayed. Under the nw (received) pane, there is a pane (called nw (sending) pane) where the message that has been just sent by a node (or just put into the network) is displayed. For each node $i = 1, 2, 3$, there are places to display the node i's $request$, j, $have\_privilege$ and rn. There is always exactly one $queue$ that is meaningful and then there is one place to display the meaningful $queue$. There is always exactly one ln that is meaningful and then there is one place to display the meaningful ln. If there is a node whose $have\_privilege$ is true, its $queue$ and ln are displayed there. If there is

[1] In general, a matching equation is in the form $T1$:= $T2$, where $T1$ and $T2$ are terms.



nw
msg(1, req(2, 5))

tran: sendReq(2)

nw (received)

N1 request: true
Idx: 1
havePrillege: true
RN (1 : 4), (2 : 4), 3 : 4

N2 request: true
Idx: 3
havePrillege: false
RN (1 : 4), (2 : 5), 3 : 4

N3 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 4), 3 : 4

nw (sending)

LN (1 : 4), (2 : 4), 3 : 3
queue 3 | empty

State 242 : (nw: msg(1, req(2, 5))) (tran: sendReq(2)) (#req[1]: 6) (#req[2]: 5) (#req[3]: 5) (pc[1]: l9) (pc[2]: l4) (pc[3]: l5) (requesting[1]: true) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: true) (havePriv[2]: false) (havePriv[3]: false) (rn[1]: (1 : 4), (2 : 4), 3 : 4) (rn[2]: (1 : 4), (2 : 5), 3 : 4) (rn[3]: (1 : 4), (2 : 4), 3 : 4) (ln[1]: (1 : 4), (2 : 4), 3 : 3) (ln[2]: (1 : 3), (2 : 4), 3 : 3) (ln[3]: (1 : 2), (2 : 3), 3 : 3) (queue[1]: 3 | empty) (queue[2]: 1 | 3 | empty) (queue[3]: 1 | 2 | empty) (idx[1]: 1) (idx[2]: 3) (idx[3]: 1)

nw
msg(1, req(2, 5)) ; msg(3, req(2, 5))

tran: sendReq(2)

nw (received)

N1 request: true
Idx: 1
havePrillege: true
RN (1 : 4), (2 : 4), 3 : 4

N2 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 5), 3 : 4

N3 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 4), 3 : 4

nw (sending)
msg(3, req(2, 5))

LN (1 : 4), (2 : 4), 3 : 3
queue 3 | empty

State 243 : (nw: msg(1, req(2, 5)) ; msg(3, req(2, 5))) (tran: sendReq(2)) (#req[1]: 6) (#req[2]: 5) (#req[3]: 5) (pc[1]: l9) (pc[2]: l5) (pc[3]: l5) (requesting[1]: true) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: true) (havePriv[2]: false) (havePriv[3]: false) (rn[1]: (1 : 4), (2 : 4), 3 : 4) (rn[2]: (1 : 4), (2 : 5), 3 : 4) (rn[3]: (1 : 4), (2 : 4), 3 : 4) (ln[1]: (1 : 4), (2 : 4), 3 : 3) (ln[2]: (1 : 3), (2 : 4), 3 : 3) (ln[3]: (1 : 2), (2 : 3), 3 : 3) (queue[1]: 3 | empty) (queue[2]: 1 | 3 | empty) (queue[3]: 1 | 2 | empty) (idx[1]: 1) (idx[2]: 1) (idx[3]: 1)

nw
msg(1, req(2, 5)) ; msg(3, req(2, 5))
msg(3, priv(empty, (1 : 4), (2 : 4), 3 : 3))

tran: transferPriv(1)

nw (received)

N1 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 4), 3 : 4

N2 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 5), 3 : 4

N3 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 4), 3 : 4

nw (sending)
msg(3, priv(empty, (1 : 4), (2 : 4), 3 : 3))

LN
queue

State 244 : (nw: msg(1, req(2, 5)) ; msg(3, req(2, 5)) ; msg(3, priv(empty, (1 : 4), (2 : 4), 3 : 3))) (tran: transferPriv(1)) (#req[1]: 6) (#req[2]: 5) (#req[3]: 5) (pc[1]: l10) (pc[2]: l5) (pc[3]: l5) (requesting[1]: true) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: false) (havePriv[2]: false) (havePriv[3]: false) (rn[1]: (1 : 4), (2 : 4), 3 : 4) (rn[2]: (1 : 4), (2 : 5), 3 : 4) (rn[3]: (1 : 4), (2 : 4), 3 : 4) (ln[1]: (1 : 4), (2 : 4), 3 : 3) (ln[2]: (1 : 3), (2 : 4), 3 : 3) (ln[3]: (1 : 2), (2 : 3), 3 : 3) (queue[1]: 3 | empty) (queue[2]: 1 | 3 | empty) (queue[3]: 1 | 2 | empty) (idx[1]: 1) (idx[2]: 1) (idx[3]: 1)

nw
msg(1, req(2, 5)) ; msg(3, req(2, 5))

tran: waitPriv(3)

nw (received)
msg(3, priv(empty, (1 : 4), (2 : 4), 3 : 3))

N1 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 4), 3 : 4

N2 request: true
Idx: 1
havePrillege: false
RN (1 : 4), (2 : 5), 3 : 4

N3 request: true
Idx: 1
havePrillege: true
RN (1 : 4), (2 : 4), 3 : 4

nw (sending)

LN (1 : 4), (2 : 4), 3 : 3
queue empty

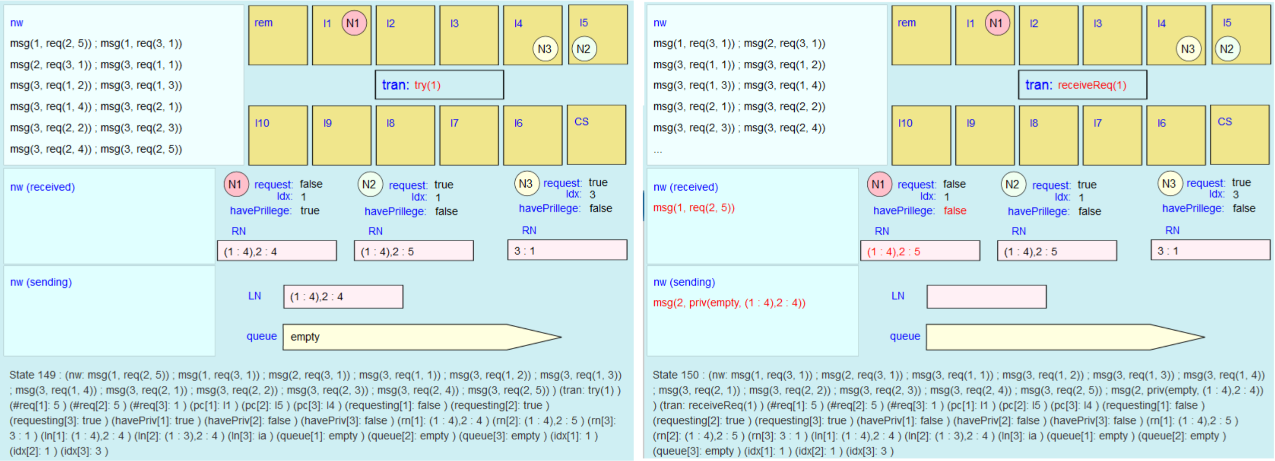
State 245 : (nw: msg(1, req(2, 5)) ; msg(3, req(2, 5))) (tran: waitPriv(3)) (#req[1]: 6) (#req[2]: 5) (#req[3]: 5) (pc[1]: l10) (pc[2]: l5) (pc[3]: cs) (requesting[1]: true) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: false) (havePriv[2]: false) (havePriv[3]: true) (rn[1]: (1 : 4), (2 : 4), 3 : 4) (rn[2]: (1 : 4), (2 : 5), 3 : 4) (rn[3]: (1 : 4), (2 : 4), 3 : 4) (ln[1]: (1 : 4), (2 : 4), 3 : 3) (ln[2]: (1 : 3), (2 : 4), 3 : 3) (ln[3]: (1 : 4), (2 : 4), 3 : 3) (queue[1]: 3 | empty) (queue[2]: 1 | 3 | empty) (queue[3]: empty) (idx[1]: 1) (idx[2]: 1) (idx[3]: 1)

Figure 6. A sequence of pictures for M_{SK} (1)

a privilege message in the network, namely that there is no node whose *have_privilege* is true, then nothing is displayed there because you can see the meaningful *queue* and ln in the privilege message in the network. There are 12 panes that correspond to the 12 locations, such as rs and l1. There is one more pane in the picture where the transition that has been just taken is displayed.

Fig. 6 shows a sequence of four pictures for M_{SK}. The four pictures correspond to four consecutive states State 242, State 243, State 244 and State 245 in a finite computation of M_{SK}. State 242 goes to State 243 by rewrite rule `sndReq` (or `sendReq(2)`), State 243 goes to State 244 by rewrite rule `trsPrv` (or `transferPriv(1)`) and State 244 goes to State 245 by rewrite rule `wtPrv` (or `waitPriv(3)`). Taking a look at the first picture (of State 242) immediately makes us recognize that node 1 is located at l9, node 2 is located at l4, node 3 is located at l5, node 1 owns the privilege, there is one message denoted `msg(1,req(2,5))`, the rule `sendReq(2)` (or `sndReq(2)`) has been just taken and so on. What is displayed as the content of ln is `(1 : 4), (2 : 4), 3 : 3`, which says that $ln[1]$ is 4, $ln[2]$ is 4 and $ln[3]$ is 3, meaning that the node 1's fourth request has been granted, the node 2's fourth request has been granted and the node 3's third request has been granted. What is displayed as the content of *queue* is `3 | empty`, which says that there is one element in *queue* and the element is 3, meaning that node 3 has been waiting to enter its critical section.

Taking a look at the second picture (of State 243) makes us recognize that the rule `sendReq(2)` (or `sndReq(2)`) has been just taken, the message `msg(1,req(2,5))` has been just put into the network and node 2 has just moved to l5 from l6. Taking a look at the third picture (of State 244) makes us recognize that the rule `transferPriv(1)` (or `trsPrv(1)`) has been just taken, the message `msg(3,priv(empty, (1 : 4),(2 : 4),3 : 4))` has been just put into the network and node 1 has just moved to l10 from l9. Taking a look at the fourth picture (of State 245) makes us recognize that the rule `waitPriv(3)` (or `wtPrv(3)`) has been just taken, the privilege message has been just received by node 3 (or just deleted from the network) and node 3 has just moved to cs from l5.



nw
msg(1, req(2, 5)) ; msg(1, req(3, 1))
msg(2, req(3, 1)) ; msg(3, req(1, 1))
msg(3, req(1, 2)) ; msg(3, req(1, 3))
msg(3, req(1, 4)) ; msg(3, req(2, 1))
msg(3, req(2, 2)) ; msg(3, req(2, 3))
msg(3, req(2, 4)) ; msg(3, req(2, 5))

tran: try(1)

N1 request: false ldx: 1 havePrillege: true RN (1 : 4),2 : 4
N2 request: true ldx: 1 havePrillege: false RN (1 : 4),2 : 5
N3 request: true ldx: 3 havePrillege: false RN 3 : 1

nw (received)

nw (sending)

LN (1 : 4),2 : 4
queue empty

State 149 : (nw: msg(1, req(2, 5)) ; msg(1, req(3, 1)) ; msg(2, req(3, 1)) ; msg(3, req(1, 1)) ; msg(3, req(1, 2)) ; msg(3, req(1, 3)) ; msg(3, req(1, 4)) ; msg(3, req(2, 1)) ; msg(3, req(2, 2)) ; msg(3, req(2, 3)) ; msg(3, req(2, 4)) ; msg(3, req(2, 5))) (tran: try(1)) (#req[1]: 5) (#req[2]: 5) (#req[3]: 1) (pc[1]: l1) (pc[2]: l5) (pc[3]: l4) (requesting[1]: false) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: true) (havePriv[2]: false) (havePriv[3]: false) (rn[1]: (1 : 4),2 : 4) (rn[2]: (1 : 4),2 : 5) (rn[3]: 3 : 1) (ln[1]: (1 : 4),2 : 4) (ln[2]: (1 : 3),2 : 4) (ln[3]: ia) (queue[1]: empty) (queue[2]: empty) (queue[3]: empty) (idx[1]: 1) (idx[2]: 1) (idx[3]: 3)

nw
msg(1, req(3, 1)) ; msg(2, req(3, 1))
msg(3, req(1, 1)) ; msg(3, req(1, 2))
msg(3, req(1, 3)) ; msg(3, req(1, 4))
msg(3, req(2, 1)) ; msg(3, req(2, 2))
msg(3, req(2, 3)) ; msg(3, req(2, 4))
...

tran: receiveReq(1)

N1 request: false ldx: 1 havePrillege: false RN (1 : 4),2 : 5
N2 request: true ldx: 1 havePrillege: false RN (1 : 4),2 : 5
N3 request: true ldx: 3 havePrillege: false RN 3 : 1

nw (received)
msg(1, req(2, 5))

nw (sending)
msg(2, priv(empty, (1 : 4),2 : 4))

LN
queue

State 150 : (nw: msg(1, req(3, 1)) ; msg(2, req(3, 1)) ; msg(3, req(1, 1)) ; msg(3, req(1, 2)) ; msg(3, req(1, 3)) ; msg(3, req(1, 4)) ; msg(3, req(2, 1)) ; msg(3, req(2, 2)) ; msg(3, req(2, 3)) ; msg(3, req(2, 4)) ; msg(3, req(2, 5)) ; msg(2, priv(empty, (1 : 4),2 : 4))) (tran: receiveReq(1)) (#req[1]: 5) (#req[2]: 5) (#req[3]: 1) (pc[1]: l1) (pc[2]: l5) (pc[3]: l4) (requesting[1]: false) (requesting[2]: true) (requesting[3]: true) (havePriv[1]: false) (havePriv[2]: false) (havePriv[3]: false) (rn[1]: (1 : 4),2 : 5) (rn[2]: (1 : 4),2 : 5) (rn[3]: 3 : 1) (ln[1]: (1 : 4),2 : 4) (ln[2]: (1 : 3),2 : 4) (ln[3]: ia) (queue[1]: empty) (queue[2]: empty) (queue[3]: empty) (idx[1]: 1) (idx[2]: 1) (idx[3]: 3)

Figure 7. A sequence of pictures for M_{SK} (2)

Fig. 7 shows another sequence of two pictures for M_{SK}. The two pictures correspond to two consecutive states State 149 and State 150 in another finite computation of M_{SK}. State 149 goes to State 150 by rewrite rule `recReq` (or `receiveReq(1)`). If it is possible to display all messages in the network on the nw pane, SMGA does so. Otherwise, a limited number of messages are displayed on the nw pane and the others are depressed. In the first picture (of State 149), all messages in the network are displayed on the nw pane. `msg(1,req(2,5))` is received by node 1 and `msg(2,priv(empty, (1 : 4),2 : 4))` is put into the network by node 1. Then, it is impossible to display all messages in the network on the nw pane. Therefore, 10 messages out of 12 ones are displayed on the nw pane and the two messages `msg(3,req(2,5))` and `msg(2,priv(empty, (1 : 4),2 : 4))` are depressed in the second picture (of State 150). Instead of displaying the two messages, "..." is displayed on the nw pane in addition to the 10 messages.

VII. Confirmation of Guessed Properties with Model Checking

Observing some graphical animations of S_{SK}, we guess that there is always at most one node at cs, l6, l7, l8 or l9 at any given moment. This is true in the six pictures shown in Fig. 6 and Fig. 7. The guessed property can be confirmed by model checking with Maude as follows:

```
search [1] in SK : ic
=>* {(pc[I]: cs) (pc[J]: cs) OCs} .
search [1] in SK : ic
=>* {(pc[I]: cs) (pc[J]: l6) OCs} .
search [1] in SK : ic
=>* {(pc[I]: cs) (pc[J]: l7) OCs} .
search [1] in SK : ic
=>* {(pc[I]: cs) (pc[J]: l8) OCs} .
search [1] in SK : ic
=>* {(pc[I]: cs) (pc[J]: l9) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l6) (pc[J]: l6) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l6) (pc[J]: l7) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l6) (pc[J]: l8) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l6) (pc[J]: l9) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l7) (pc[J]: l7) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l7) (pc[J]: l8) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l7) (pc[J]: l9) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l8) (pc[J]: l8) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l8) (pc[J]: l9) OCs} .
search [1] in SK : ic
=>* {(pc[I]: l9) (pc[J]: l9) OCs} .
```

For example, the first search command tries to find a reachable state in which there are two processes at cs simultaneously. Maude does not find any reachable states from `ic` that match any of those patterns. Therefore, the guessed property is invariant with respect to M_{SK} when there are three nodes and each node enters its critical section once.

Observing some graphical animations of S_{SK}, we also guess that there is a privilege message in the network if and only if there is no node located at cs, l6, l7, l8 or l9 at any given moment. This is true in the six pictures shown in Fig. 6 and Fig. 7. The guessed property can be confirmed by model checking with Maude as follows:

```
search [1] in SK : ic
=>* {(nw: (msg(I,priv(Q,A)) ; NW))
       (pc[J]: cs) OCs} .
search [1] in SK : ic
=>* {(nw: (msg(I,priv(Q,A)) ; NW))
```

```
          (pc[J]: l6) OCs} .
search [1] in SK : ic
=>* {(nw: (msg(I,priv(Q,A)) ; NW))
          (pc[J]: l7) OCs} .
search [1] in SK : ic
=>* {(nw: (msg(I,priv(Q,A)) ; NW))
          (pc[J]: l8) OCs} .
search [1] in SK : ic
=>* {(nw: (msg(I,priv(Q,A)) ; NW))
          (pc[J]: l9) OCs} .
```

For example, the first search command tries to find a reachable state in which there is a privilege message in the network and a process is located at cs simultaneously. Maude does not find any reachable states from `ic` that match any of those patterns. Therefore, the second guessed property is also invariant with respect to M_{SK} when there are three nodes and each node enters its critical section once.

Observing some graphical animations of S_{SK}, we also guess that there exists a node whose *have_privilege* is true if and only if there does not exist any other node at cs, l6, l7, l8 or l9 at any given moment. This is true in the six pictures shown in Fig. 6 and Fig. 7. The guessed property can be confirmed by model checking with Maude as follows:

```
search [1] in SK : ic
=>* {(pc[I]: cs) (havePriv[J]: true) OCs}
such that I =/= J .
search [1] in SK : ic
=>* {(pc[I]: l6) (havePriv[J]: true) OCs}
such that I =/= J .
search [1] in SK : ic
=>* {(pc[I]: l7) (havePriv[J]: true) OCs}
such that I =/= J .
search [1] in SK : ic
=>* {(pc[I]: l8) (havePriv[J]: true) OCs}
such that I =/= J .
search [1] in SK : ic
=>* {(pc[I]: l9) (havePriv[J]: true) OCs}
such that I =/= J .
```

For example, the first search command tries to find a reachable state in which a process is located at cs and some other process *have_privilege* is true simultaneously. Maude does not find any reachable states from `ic` that match any of those patterns. Therefore, the third guessed property is also invariant with respect to M_{SK} when there are three nodes and each node enters its critical section once.

VIII. Related Work

A graphical user interface for Maude-NPA has been developed [8]. Maude-NPA is a high-level security protocol analysis language and system implemented on the top of Maude. The graphical user interface is dedicated to Maude-NPA and then cannot be used for our purpose, namely that graphical animations of state machines can be displayed, helping humans guess properties of the state machines based on the graphical animations.

Specification animation has been actively studied. Specification animation means making formal specification executable by translating formal specifications into executable programs because most formal specification languages are not executable. Specification animations have been used to inspect formal specifications [9], to monitor software through formal specification animation [10], to validate formal models by refinement animation [11] and to make a specification-based testing better [12]. Maude is inherently executable and then it is unnecessary to translate Maude specifications into executable programs.

May Thu Aung, et al. [13] uses a mutual exclusion protocol called Qlock to conduct a case study in which some properties of Qlock have been guessed by observing graphical animations of Qlock generated with SMGA, model checked with Maude and theorem proved with CafeOBJ [14]. One piece of our future work is to theorem prove the properties guessed in this paper with CafeOBJ.

Computer networks have been grown and intricate. Social networking service (SNS) has been used by many people over the world and then networks constituted of those SNS users have become very complex. Visualization is one promising way to comprehend such complex networks and then network visualization has been intensively studied. Tools, such as Gephi [15] and Cytoscape [16], have been developed. Some visualization techniques used in those tools could be used for our purpose. Because state machines cannot be necessarily expressed as networks, however, those network visualization tools cannot be directly used to graphically animate state machines.

IX. Conclusion

We have described graphical animations of the Suzuki-Kasami protocol with SMGA. Observing them has made us guess some properties of the state machine formalizing the Suzuki-Kasami protocol. We have used the Maude reachability analyzer (the search command) to confirm that the guessed properties are invariant with respect to the state machine when there are three nodes and each node enters its critical section once. The case study demonstrates that state machine graphical animations could make humans perceive state machine properties. Another piece of our future work is to generate graphical animations of some other distributed protocols, such as Paxos [17], guess some properties of those protocols by observing the graphical animations, confirm the guessed properties by model checking and theorem prove them.

References

[1] T. T. T. Nguyen and K. Ogata, “Graphical animations of state machines,” in *15th IEEE Intl Conf. Depend., Auton. & Secure Compu.*, 2017, pp. 604–611.

[2] I. Suzuki and T. Kasami, “A distributed mutual exclusion algorithm,” *ACM Trans. Comput. Syst.*, vol. 3, no. 4, pp. 344–349, 1985.

[3] T. T. T. Nguyen and K. Ogata, "Graphically perceiving characteristics of the MCS lock and model checking them," in *7th Intl Wksh SOFL+MSVL*, ser. Lecture Notes in Computer Science, vol. 10795. Springer, 2017, pp. 3–23.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[5] K. Ogata and K. Futatsugi, "Formal analysis of Suzuki & Kasami distributed mutual exclusion algorithm," in *5th Intl Conf. Formal Methods for Open Object-Based Dist. Sys.*, ser. IFIP Conference Proceedings, vol. 209. Kluwer, 2002, pp. 181–195.

[6] ——, "Analysis of the Suzuki-Kasami algorithm with the Maude model checker," in *12th Asia-Pacific Soft. Eng. Conf.* IEEE Computer Society, 2005, pp. 159–166.

[7] ——, "Comparison of Maude and SAL by conducting case studies model checking a distributed algorithm," *IEICE Trans.*, vol. 90-A, no. 8, pp. 1690–1703, 2007.

[8] S. Santiago, C. L. Talcott, S. Escobar, C. A. Meadows, and J. Meseguer, "A graphical user interface for Maude-NPA," in *9th Spanish Conf. Prog. & Lang.*, 2009, pp. 3–20.

[9] M. Li and S. Liu, "Integrating animation-based inspection into formal design specification construction for reliable software systems," *IEEE Trans. Reliability*, vol. 65, no. 1, pp. 88–106, 2016.

[10] H. Liang, J. S. Dong, J. Sun, and W. E. Wong, "Software monitoring through formal specification animation," *Innov. in Sys. & Soft. Eng.*, vol. 5, no. 4, pp. 231–241, 2009.

[11] S. Hallerstede, M. Leuschel, and D. Plagge, "Validation of formal models by refinement animation," *Sci. Comput. Program.*, vol. 78, no. 3, pp. 272–292, 2013.

[12] F. Nagoya and S. Liu, "A comparative study of a GUI-aided formal specification construction approach," in *17th Intl Conf. Comput. Sci. & App.*, ser. LNCS, vol. 10404. Springer, 2017, pp. 273–283.

[13] M. T. Aung, T. T. T. Nguyen, and K. Ogata, "Guessing, model checking and theorem proving of state machine properties – a case study on Qlock," *Intl J. Softw. Eng. & Comput. Sys.*, vol. 4, no. 2, pp. 1–18, 2018.

[14] R. Diaconescu and K. Futatsugi, *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, ser. AMAST Series in Computing. World Scientific, 1998, vol. 6.

[15] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," in *3rd International Conference on Weblogs and Social Media*.

[16] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: A software environment for integratedmodels of biomolecular interaction networks," *Genome Res.*, vol. 13, no. 11, pp. 2498–2504, 2003.

[17] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.