JVLC

Journal of Visual Language and Computing

Volume 2019, Number 2

Journal of Visual Language and Computing

journal homepage: www.ksiresearch.org/jvlc

Graphical Animations of the Suzuki-Kasami Distributed Mutual Exclusion Protocol[⋆,⋆⋆]

Dang Duy Bui[a], Kazuhiro Ogata[a,*]

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

ARTICLE INFO

Article History:
Submitted 3.1.2019
Revised 6.1.2019
Second Revision 8.1.2019
Accepted 10.10.2019

Keywords:
distributed protocols
graphical animations
Maude
model checking
SMGA
state machines

ABSTRACT

A state machine that formalizes a distributed mutual exclusion protocol called the Suzuki-Kasami protocol is graphically animated. The messages that have been just put into (or sent) and deleted from (or received) the network are crucial information and then visually explicitly displayed on the designated places in a state picture. The protocol uses some pieces of information that are seemingly owned by each node but actually shared by all nodes. The pieces of information are visually explicitly displayed on two designated places. One main purpose of graphically animating state machines is to make it possible for humans to visually perceive characteristics or properties of the state machines. We demonstrate that carefully observing graphical animations makes it possible for human users to perceive some characteristics or properties of the state machine formalizing the Suzuki-Kasami protocol and the properties are confirmed by model checking. To make it more likely for human users to be able to perceive such properties, it is necessary to design good state pictures. We summarize some tips on how to design good state pictures for mutual exclusion protocols.

1. 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, and one promising way to do so would be to rely on human visual perception. Therefore, Nguyen and Ogata [8] 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 [14] is tackled with SMGA.

As all other distributed systems, the Suzuki-Kasami protocol uses a network with which messages are exchanged among nodes. 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. There may be too many messages in the network to display all of them on the designated place. If that is the case, a limited number of messages is displayed and the others are suppressed. Instead, "..." is displayed. The protocol uses some pieces of information that are seemingly owned by each node but actually shared by all nodes. The pieces of information are visually explicitly displayed on the designated places in a state picture.

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. One

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

⋆⋆The present paper is an extended and revised version of the paper [4] presented at DMSVIVA 2019.

*Corresponding author
bddang@jaist.ac.jp (D.D. Bui); ogata@jaist.ac.jp (K. Ogata)
ORCID(s): 0000-0002-2700-1762 (D.D. Bui); 0000-0002-4441-3259 (K. Ogata)

DOI reference number: 10-18293/JVLC2019-N2-012

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. 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 [5], a rewriting logic-based computer language, as a specification language for state machines and a model checker. We mention some SMGA functionalities, such as playing forward and backward frame-by-frame playback and finding out states that satisfy some conditions from a state sequence used in an input file. We also summarize how to design state pictures for mutual exclusion protocols.

The rest of the paper is organized as follows. Sect. 2 mentions some preliminaries needed to comprehend the technical contents of the paper. Sect. 3 describes state machine graphical animation (SMGA). Sect. 4 describes the Suzuki-Kasami protocol. Sect. 5 describes how to specify the Suzuki-Kasami protocol in Maude. Sect. 6 describes how to revise SMGA so that the Suzuki-Kasami protocol can be tackled and reports on the case study in which the protocol has been graphically animated with the revised version of SMGA. Sect. 7 mentions some functionalities of SMGA. Sect. 8 mentions on how to confirm guessed properties of the protocol with model checking. Sect. 9 summarizes some tips on how to design state pictures for mutual exclusion protocols. Sect. 10 mentions some related work. Sect. 11 concludes the paper and mentions some future directions.

2. Preliminaries

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;
```

Multiple processes participate in the protocol. Each process is in Remainder Section, working on some tasks that do not require any shared resources. When a process needs some shared resources, it is supposed to move to Critical Section where it uses the shared resources. After that, it goes back to Remainder Section. We suppose that each process is located at either rs or cs. A process is located at rs and cs if and only if it is in Remainder Section and Critical Section, respectively. *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*. 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 Critical Section, it sets *locked* to false and goes back to rs.

The protocol is formalized as a state machine $M \triangleq \langle S, I, T \rangle$ that consists of a set S of states, where some states $I\ \subseteq$ are initial states, and a binary relation T over states, where each element $(s, s') \in T$ is a state transition, saying that state s can go to state s'. We use Maude [5], a programming/specification language based on rewriting logic, to specify state machines. Maude makes it possible to specify complex systems flexibly and is also equipped with model checking facilities, such as a reachability analyzer (or a search command).

When there are three processes p1, p2 and p3 participating in the protocol, let $M_{\mathrm{TS}} \triangleq \langle S_{\mathrm{TS}}, I_{\mathrm{TS}}, T_{\mathrm{TS}} \rangle$ be the state machine formalizing the protocol. A state in S_{TS} can be 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. (pc[pi]: li) and (locked: b) are called observable components that are name-value pairs, meaning that process pi is located at li and the value stored in variable *locked* is b, respectively. pc[pi] and locked are names, while li and b are values. (pc[pi]: li) is also called a pc or pc[pi] observable component and (locked: b) is also called a locked observable component. Observable components are glued as members of an associative-commutative collection that is called a soup. Therefore, a state is expressed as a braced soup of observable components. 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:

```
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.

By substituting B, I and OCs with true, p3 and (pc[p1]: rs) verb!(pc[p2]: rs)!, the left-hand side of rule exit comes to equal State 61 shown in Fig. 1 because soups are associative and commutative and then the right-hand side comes to equal State 61. By substituting B, I and ICs with false, p1 and (pc[p2]: rs) (pc[p3]: rs), the left-hand side of rule enter comes to equal State 64 shown in Fig. 1 and then the right-hand side comes to equal State 65. These are how rewrite rules describe state transitions.

3. State Machine Graphical Animation (SMGA)

SMGA [8] has been implemented with Draw SVG (www.drawsvg.org). It basically takes a finite computation (or a finite state sequence) 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

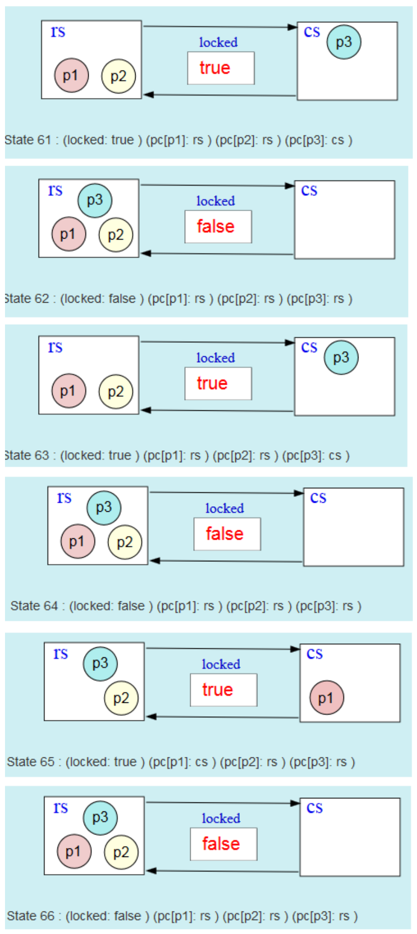

State 61 : (locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs)
State 62 : (locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
State 63 : (locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs)
State 64 : (locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
State 65 : (locked: true) (pc[p1]: cs)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

Figure 1: A sequence of pictures for M_{TS}

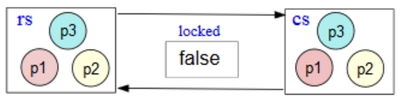


Figure 2: A picture of states in S_{TS}

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. 2 when there are three processes. An input file to SMGA consists of three parts: `###keys`, `###textDisplay` and `###states`. Fig. 3 shows a snip of an input file to SMGA for M_{TS}. In the `###keys` part, the names (or keys) used in the observable components that constitute each state are enumerated. For M_{TS}, `locked`, `pc[p1]`, `pc[p2]` and `pc[p3]` are enumerated. There are two different ways to display the value stored in an observable component: (1) the value is displayed at a designated place and (2) one of the diagram objects associated with the value appears, while the other diagram objects disappear. For (1), the value is treated as a text and displayed horizontally from left to right by default. Some data structures, such as stacks, should be displayed in a different way.

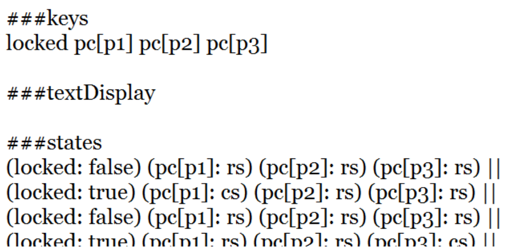

```
###keys
locked pc[p1] pc[p2] pc[p3]

###textDisplay

###states
(locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) ||
(locked: true) (pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs) ||
(locked: false) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) ||
(locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs) ||
```

Figure 3: A snip of an input file to SMGA for S_{TS}

For example, a stack should be displayed vertically from top to bottom. In the `###textDisplay` part, we can specify how values treated as texts are displayed for (1). For M_{TS}, the value stored in the `locked` observable component is displayed as (1). Because it is displayed by default, nothing is specified in the `###textDisplay` part. The values stored in the `pc[p1]`, `pc[p2]` and `pc[p3]` observable components are displayed as (2). For example, if the value stored in the `pc[p1]` observable component is `rs`, the circle on which 1 is written appears in the `rs` region and otherwise the circle appears in the `cs` region. In the `###states` part, a sequence of states is written, where states are separated by `||`.

Fig. 1 shows a sequence of pictures for M_{TS} generated by SMGA. In State 61, the value stored in the `locked` observable component is `true`, the value stored in the `pc[1]` observable component is `rs`, the value stored in the `pc[2]` observable component is `rs`, the value stored in the `pc[3]` observable component is `cs`. Hence, `true` is displayed by default at the designated place, the circle on which 1 is written appears in the `rs` region, the circle on which 2 is written appears in the `rs` region and the circle on which 3 is written appears in the `cs` region in State 61. In State 62, the value stored in the `locked` observable component changes to `false`, the value stored in the `pc[3]` observable component changes to `rs` and the other values do not change. Hence, `false` is displayed by default at the designated place, the circle on which 3 is written appears in the `rs` region and the other two circles remain in the `rs` region in State 62.

Observing such a sequence of pictures or a graphical animation of a state machine, we could guess some properties of the state machine [9]. Observing the sequence of pictures shown in Fig. 1, 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. We use the Maude reachability analyzer (or the search command) to confirm that the guessed property is invariant with respect to M_{TS}. The following command can be used to confirm it:

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

Maude exhaustively searches the reachable states from the initial state `ic` for a state that can match `{(locked: false) (pc[I]: cs) OCs}`. If there is such a state, in which a process `I` is located at cs and *locked* is false, then the property is

```
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
rstReq(i)    ⟷  l10    request := false;
                     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 4: Suzuki-Kasami distributed mutual exclusion protocol in an Algol-like language

violated. No such a state is found by Maude and then the guessed property is invariant with respect to M_{TS} when there are three processes.

4. Suzuki-Kasami Protocol

The Suzuki-Kasami protocol is a distributed mutual exclusion protocol [14]. The basic idea is that a node that has a privilege is only allowed to enter its critical section, 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 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 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 $i \in$ Node. The procedures for node i are shown in Fig. 4.

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 $j \in$ 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 $i \in$ 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 $i \in$ 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 $j \in$ 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 $privilege(\text{deq}(queue), ln)$ 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.

5. 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 $i \in$ Node by node $j \in$ Node is expressed as $\text{msg}(i, \text{req}(j, k))$, where $k \in$ Nat, msg is used as the constructor of messages and req is used as the constructor of requests. A privilege message addressed to node $i \in$ Node is expressed as $\text{msg}(i, \text{priv}(q, a))$, where $q \in$ NodeQueue, $a \in$ NatNArray and priv is used as the constructor of privileges. Let Req and Priv be $\{\text{req}(i, n) \mid i \in \text{Node}, n \in \text{Nat}\}$

```
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 [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} .

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 5: Rewrite rules specifying T_{SK}

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 $i \in$ Node, $r \in$ Req, and $p \in$ Priv, $\mathrm{msg}(i,r) \in$ MsgSoup and $\mathrm{msg}(i,p) \in$ MsgSoup, and for each $ms_1, ms_2 \in$ MsgSoup, ms_1 ; $ms_2 \in$ 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 ; void $=$ void ; $ms = ms$ for each $ms \in$ 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 $M_{\mathrm{SK}} \triangleq \langle S_{\mathrm{SK}}, I_{\mathrm{SK}}, T_{\mathrm{SK}} \rangle$, which is specified in Maude. P1 is divided into 12 regions shown in Fig. 4. 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 leftmost column in Fig. 4. 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 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)
```

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)}`, 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. There are 13 rules, among which four rules are shown in Fig. 5. 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[J] := Max` is the array assignment at index `J`. `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. The rules will be described later by observing some state pictures generated by SMGA. Let us, however, mention one thing about rule `recReq` that formalizes procedure P2. The rules has `L =/= l10 /\ L =/= l8 /\ L =/= l7` as part of the condition, saying that node `I` does not receive any requests messages if it is located at l10, l8 or l7. The condition is not explicitly mentioned in the original paper [14] of the Suzuki-Kasami protocol. If we do not use the condition, the protocol may cause lockout that a node that wants to enter its critical section never be there if a node does not try to enter its critical section unboundedly many times [10].

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

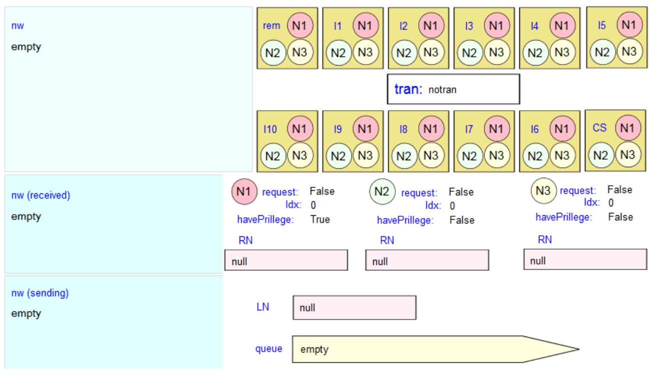
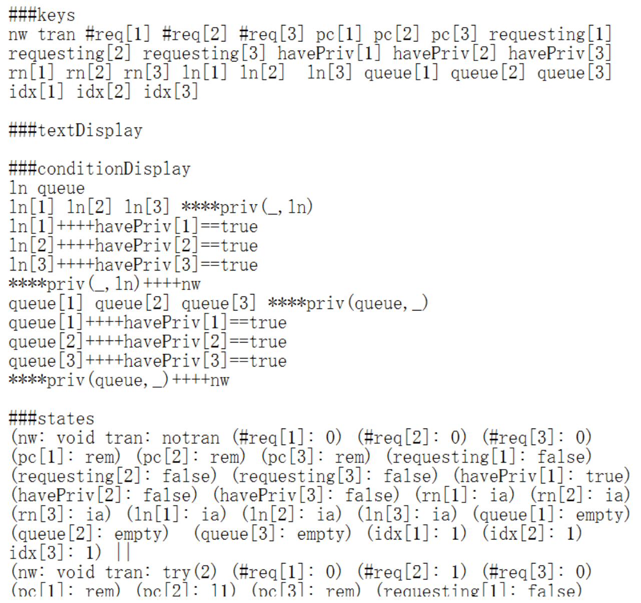


Figure 6: A picture of states in S_{SK}



```
###keys
nw tran #req[1] #req[2] #req[3] pc[1] pc[2] pc[3] requesting[1]
requesting[2] requesting[3] havePriv[1] havePriv[2] havePriv[3]
rn[1] rn[2] rn[3] ln[1] ln[2]  ln[3] queue[1] queue[2] queue[3]
idx[1] idx[2] idx[3]

###textDisplay

###conditionDisplay
ln queue
ln[1] ln[2] ln[3] ****priv(_, ln)
ln[1]++++havePriv[1]==true
ln[2]++++havePriv[2]==true
ln[3]++++havePriv[3]==true
****priv(_, ln)++++nw
queue[1] queue[2] queue[3] ****priv(queue,_)
queue[1]++++havePriv[1]==true
queue[2]++++havePriv[2]==true
queue[3]++++havePriv[3]==true
****priv(queue,_)++++nw

###states
(nw: void tran: notran (#req[1]: 0) (#req[2]: 0) (#req[3]: 0)
(pc[1]: rem) (pc[2]: rem) (pc[3]: rem) (requesting[1]: false)
(requesting[2]: false) (requesting[3]: false) (havePriv[1]: true)
(havePriv[2]: false) (havePriv[3]: false) (rn[1]: ia) (rn[2]: ia)
(rn[3]: ia) (ln[1]: ia) (ln[2]: ia) (ln[3]: ia) (queue[1]: empty)
(queue[2]: empty)  (queue[3]: empty) (idx[1]: 1) (idx[2]: 1)
idx[3]: 1) ||
(nw: void tran: try(2) (#req[1]: 0) (#req[2]: 1) (#req[3]: 0)
(pc[1]: rem) (pc[2]: l1) (pc[3]: rem) (requesting[1]: false)
```

Figure 7: A snip of an input file to SMGA for S_{SK}

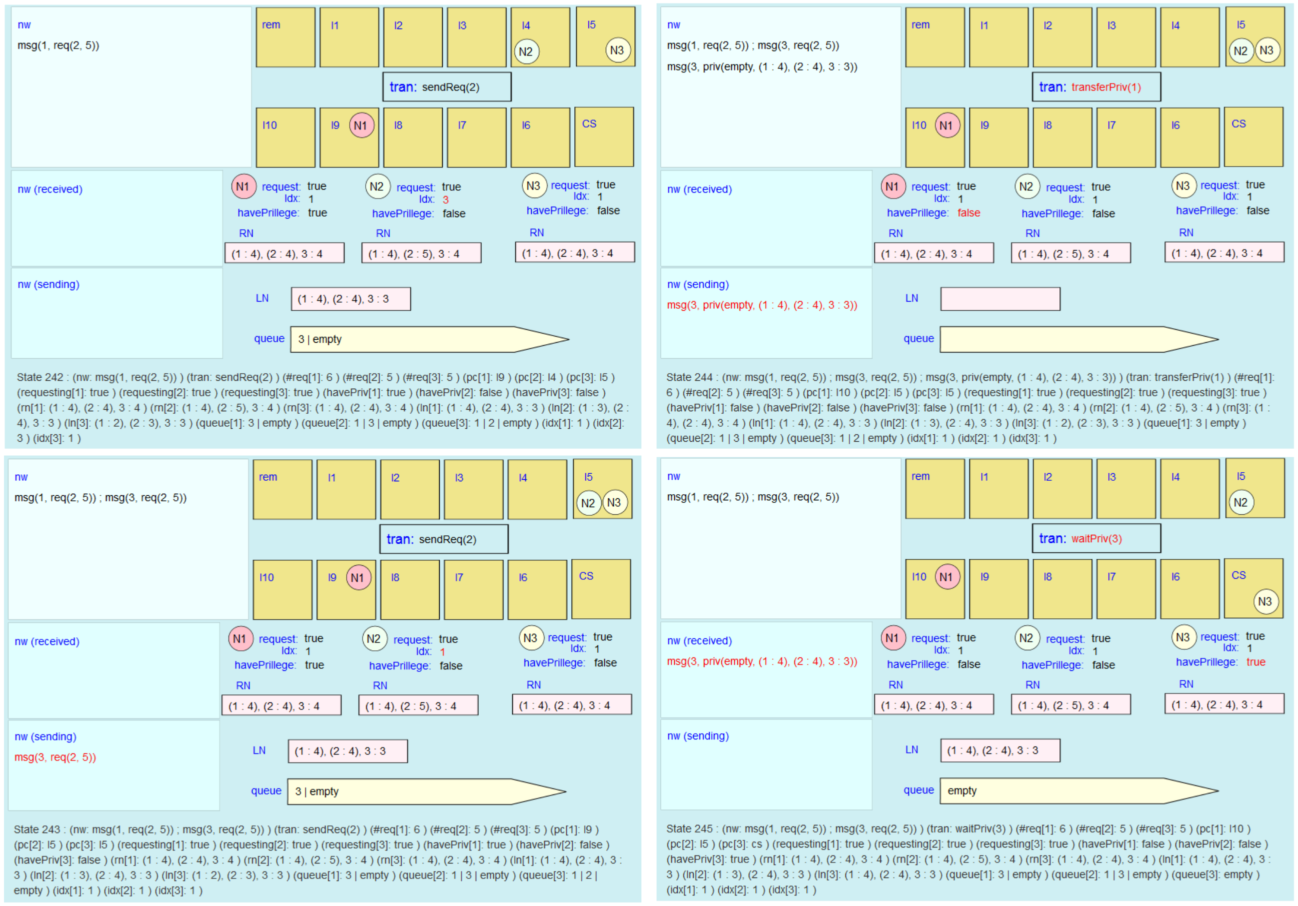
6. Graphical Animations of Suzuki-Kasami Protocol

Fig. 6 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 (or `idx`), *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 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. 7 shows a snip of an input file to SMGA for M_{SK}. We have added one more part called the `###conditionDisplay` part to an input file to SMGA. Although each node has its own variables ln and *queue*, they have meaningful values only if the node owns the privilege. Hence, we would like to only display the meaningful ln and *queue* on the two places on a state picture, respectively. Just below the line where `###conditionDisplay` is written in the input file shown in Fig. 7, we write the two names `ln` and `queue` because at most one `ln` and at most one `queue` are displayed. The next line `ln[1] ln[2] ln[3] ****priv(_,ln)` specifies that there are four possible places where the meaningful `ln` is stored: the `ln[1]` observable component, the `ln[2]` observable component, the `ln[3]` observable component and the privilege message in the network. The next line `ln[1]++++havePriv[1]==true` says that if the `havePriv[1]` observable component stores `true`, the value stored in the `ln[1]` observable component is displayed on the place for ln. The following two lines can be interpreted likewise. The next line `****priv(_,ln)++++nw` says that if there exists a privilege message in the network, nothing is displayed on the place for ln. The following five lines specify how to deal with *queue* and can be interpreted in the same way as the five lines for ln.

The `nw` observable component consists of the messages that have been sent and have not been received yet. Those messages are displayed on the nw pane. It may be impossible to display all messages on the nw pane. If that is the case, the messages that can fit to the nw pane are only displayed, which are followed by "..." If there exists a message that has been just put (or sent) into the network, the message is displayed on the nw (sending) pane. If there exists a message that has been just deleted (or received) from the network, the message is displayed on the nw (received) pane.

Fig. 8 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 re-



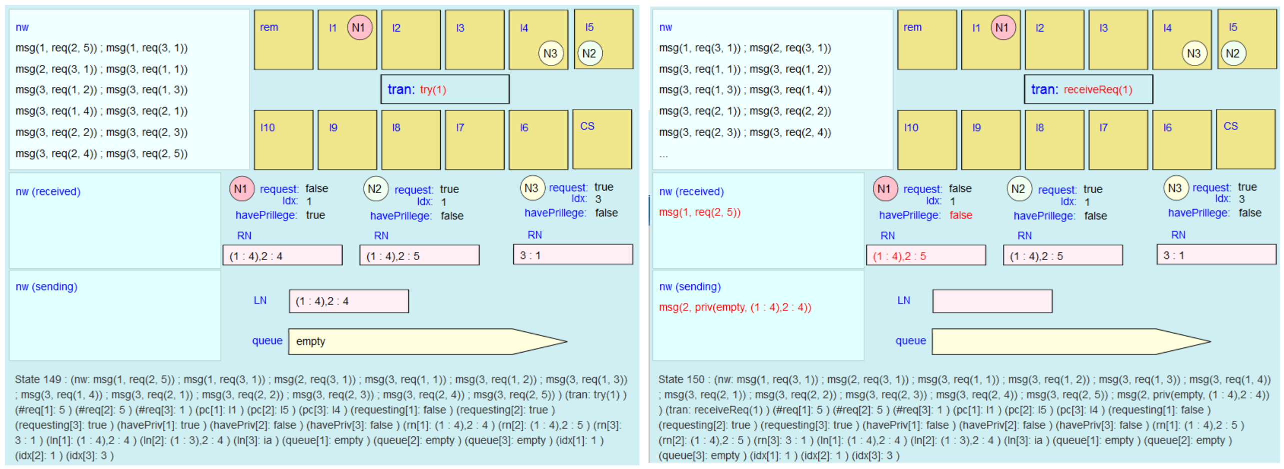
```
nw
msg(1, req(2, 5))

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))
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: 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))

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 8: A sequence of pictures for M_{SK} (1)



```
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)

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

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)

nw (received)
msg(1, req(2, 5))
N1 request: false  idx: 1  havePrillege: false  RN (1 : 4),2 : 5
N2 request: true   idx: 1  havePrillege: false  RN (1 : 4),2 : 5
N3 request: true   idx: 3  havePrillege: false  RN 3 : 1

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 9: A sequence of pictures for M_{SK} (2)

quest 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. The state transition from State 242 to State 243 visually describes an instance of what rule `sndReq` (or `sendReq(2)` or `sndReq(2)`) in Fig. 5 does.

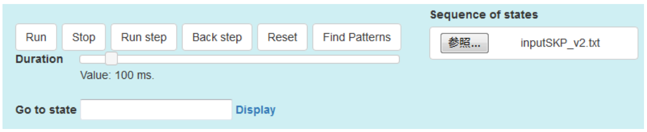
Taking a look at the third picture (of State 244) makes us recognize that the rule `transferPriv(1)` (or `trsPrv(1)`) has been just taken, the message `msg(3,priv(empty, (1 : 4),(2 : 4),3 : 4))` has been just put into the network and node 1 has just moved to l10 from l9. The state transition from State 243 to State 244 visually describes an instance of what rule `trsPrv` (or `transferPriv(1)` or `trsPrv(1)`) in Fig. 5 does.

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. The state transition from State 244 to State 245 visually describes an instance of what rule `wtPrv` (or `transferPriv(1)` or `trsPrv(1)`) in Fig. 5 does.

Fig. 9 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.

7. Some Functionalities of SMGA

In addition to screening state machine movies, SMGA allows us to play forward and backward frame-by-frame playback, etc. Under the state sequence movie (or state picture) displayed by SMGA, there are several buttons shown as follows:

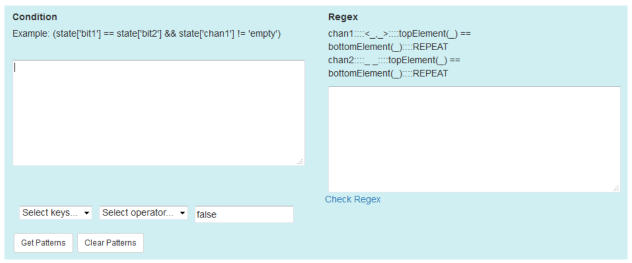


Value: 100 ms.

inputSKP_v2.txt

The "Run step" button plays forward frame-by-frame playback and the "Back step" does backward one. The "Duration" adjuster can change how fast a movie is played. When you write a number N in the box just right-hand side of "Go to state" and click "Display," the picture of state N is displayed.

The "Find Patterns" button makes it possible to find states that satisfy some conditions. When the button is clicked, the following appears:



Example: (state['bit1'] == state['bit2'] && state['chan1'] != 'empty')

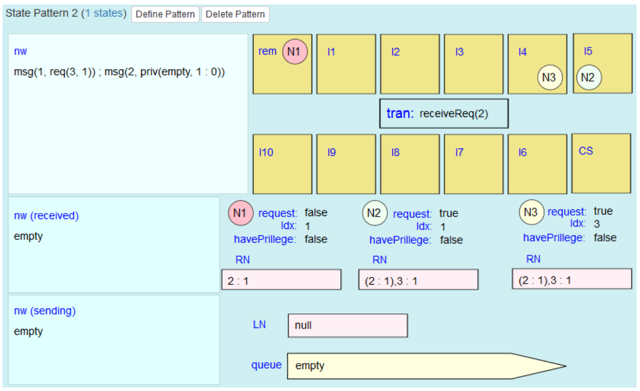
chan1::::<_,_>::::topElement(_) ==
bottomElement(_)::::REPEAT
chan2::::_ _::::topElement(_) ==
bottomElement(_)::::REPEAT

false

Writing conditions in the box under "Condition" and regular expressions in the box under "Regex", the "Get Patterns" button finds all states appearing in the state sequence of the input file that satisfy the conditions and regular expressions. When the following condition is written in the condition box,

```
(state['havePriv[1]'] == 'false' ) &&
(state['havePriv[2]'] == 'false' ) &&
(state['havePriv[3]'] == 'false' )
```

SMGA finds out 7 states among 1000 states that satisfy the condition in the first input file. One state found is as follows:



msg(1, req(3, 1)) ; msg(2, priv(empty, 1 : 0))

tran: receiveReq(2)

nw (received) empty

N1 request: false Idx: 1 havePrillege: false RN 2 : 1

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

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

nw (sending) empty

LN null

queue empty

We notice that there exists a privilege message in the network in the state.

When the following condition is written in the condition box,

```
(state['pc[1]'] == 'cs') &&
(state['pc[2]'] == 'cs')
```

SMGA does not find out any states among 1000 states that satisfy the condition in the first input file. From it, we may guess that two or more nodes can not be in their critical sections at the same time. For the following condition,

```
(state['havePriv[1]']  ==  'true') &&
(state['havePriv[2]']  ==  'true')
```

SMGA does find out any states, either. From it, we may guess that two or more node *have_privilege*'s cannot be true at the same time.

The SMGA functionality that finds out states that match regular expression for the Suzuki-Kasami protocol has not been implemented. We would like to express some regular expression that says that there exists a privilege message in the network. It is one piece of our future work to implement the SMGA functionality with regular expressions.

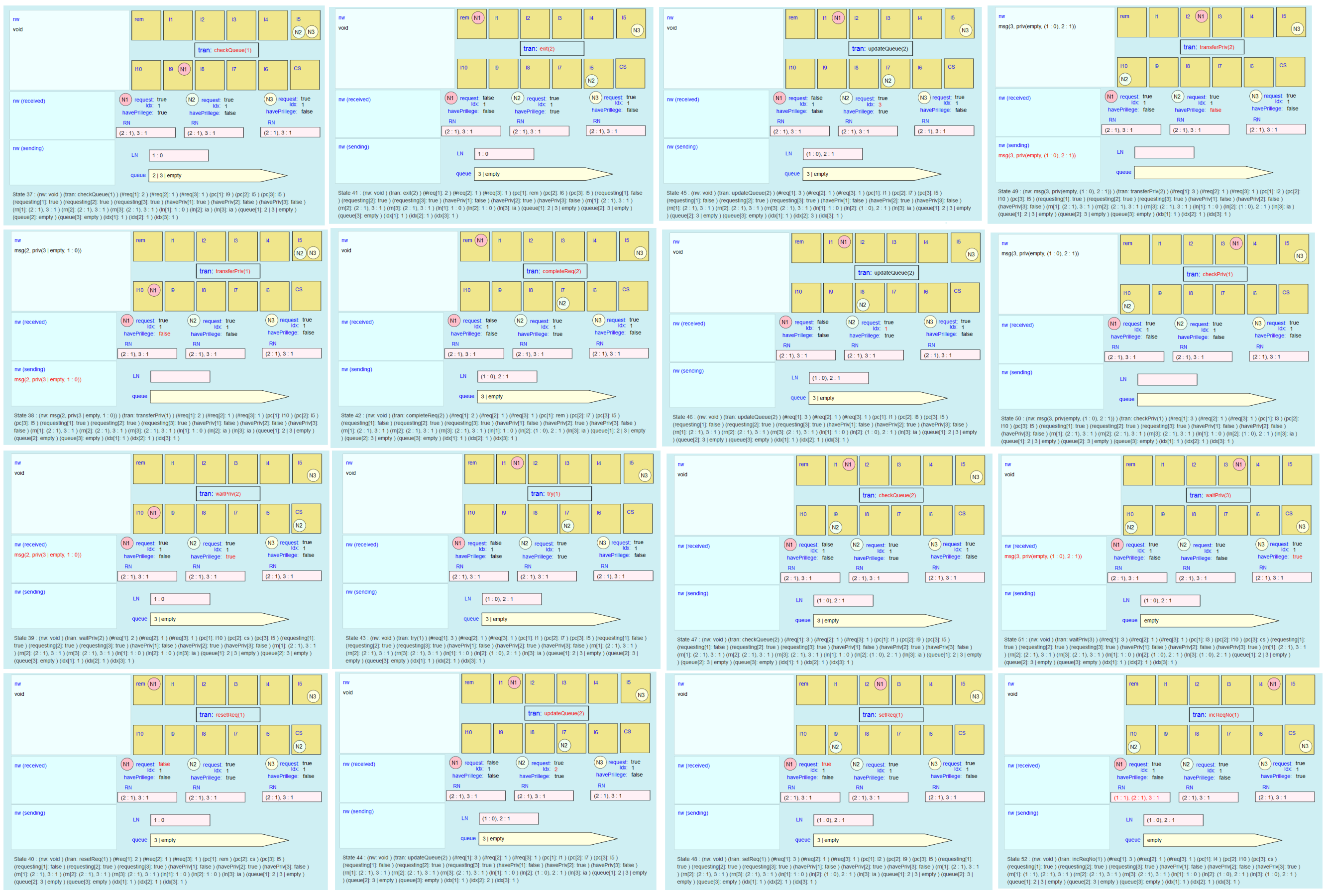


Figure 10: A sequence of pictures for M_{SK} (3)

8. Confirmation of Guessed Properties with Model Checking

Observing some graphical animations of M_{SK} could help us visually perceive some characteristics or properties of M_{SK}. Fig. 10 shows yet another sequence of states for M_{SK}. Carefully observing such graphical animations of M_{SK}, we notice that there is always at most one node located at cs, l6, l7, l8 or l9 at any given moment. The guessed property can be confirmed by model checking with Maude as follows:

```
search [1] in SK : ic
=>* (pc[I]: l1) (pc[J]: l2) OCs .
```

for $l1, l2 \in \{\mathrm{cs}, l6, l7, l8, l9\}$. The search command checks if there is a state reachable from `ic` such that two nodes `I` & `J` are located at cs, l6, l7, l8 or l9 at the same time. Because the command does not find any such state, the guessed property is true when there are three nodes.

Carefully observing some graphical animations of M_{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 all pictures shown in Fig. 8, Fig. 9 and Fig. 10. 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]: l1) OCs .
```

for $l1 \in \{\mathrm{cs}, l6, l7, l8, l9\}$. The search command checks if there is a state reachable from `ic` such that a privilege message is in the network and a node is located at cs, l6, l7, l8 or l9 at the same time. Because the command does not find any such state, the guessed property is true when there are three nodes.

Carefully 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 also true in all pictures shown in Fig. 8, Fig. 9 and Fig. 10. The guessed property can be confirmed by model checking with Maude as follows:

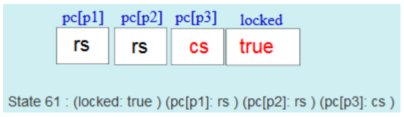
```
search [1] in SK : ic
=>* (pc[I]: l1) (havePriv[J]: true) OCs
such that I =/= J .
```

for $l1 \in \{\mathrm{cs}, l6, l7, l8, l9\}$. The search command checks if there is a state reachable from `ic` such that a node `I` is located at cs, l6, l7, l8 or l9 and a node `j` owns the privilege at the same time, where `I`≠`J`. Because the command does not find any such state, the guessed property is true when there are three nodes.

Note that model checking only guarantees that the three guessed properties are true when there are three nodes. We need to use theorem proving so as to guarantee that the properties are true when there are an arbitrary number of nodes [2].

9. Some Tips on How to Design State Pictures

SMGA does not automatically produce visual representations or pictures of states but human users are supposed to design state pictures. Any state pictures are not good. For example, one possible state picture of the test&set protocol is as follows:



State 61 : (locked: true) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: cs)

Although state pictures like this could be automatically generated, we do not think that this state picture is very good for the test&set protocol because the state picture is almost the same as the text representation shown below the state picture. The state picture for the test&set protocol shown in Fig. 1 and Fig. 2 is better than the one shown above. This is because (1) we can immediately realize how many sections there are totally, (2) how many processes there are in each section and (3) the relations among the sections such that a process located at cs will move to rs. We can also visually perceive that (4) variable *locked* is shared by all processes and (5) what section each process is located at.

We summarize some tips on how to design state pictures for mutual exclusion protocols based on our experiences.

- To recognize what sections there are at which each process or node is located, allocate the pane (or place) for each section such that the relations among the sections are visually perceived and display some diagram, such as a circle on which a process or node ID is written, on the designated pane;

- To recognize what pieces of information, such as the network for the Suzuki-Kasami protocol and variable *locked* for the test&set protocol, are shared by all processes or nodes, allocate the pane (or place) for each such piece of information such that we can visually perceive they are shared by all processes and nodes and display them on the designated panes adequately;

- To recognize whether there are some that are more crucial than the others among the shared resources, such as the messages that have been just put into and deleted from the network, prepare the panes (or places) for them and display them there adequately;

- To recognize what pieces of information are owned by each process or node, allocate the panes (or places) for them to make it possible to visually perceive what pieces of information are owned by what processes or nodes and display them on the designated panes adequately.

There may be some pieces of information that are seemingly stored in each process or node variables but actually the pieces of information are shared by all processes or nodes. *queue* and *ln* used in the Suzuki-Kasami protocol are such pieces of information. To realize it, we need to comprehend the Suzuki-Kasami protocol to some extent. One important lesson learned from our experiences is that it is necessary to comprehend mutual exclusion protocols well to some extent so as to design reasonably good state pictures.

10. Related Work

Alloy [7] is a relational-logic based specification (or modeling) language. Its environment is also called Alloy. Alloy is equipped with a SAT-based bounded model checker. When it finds a counterexample, it automatically visualizes states. PAT (Process Analysis Toolkit) [13] is an enhanced simulator, model checker and refinement checker for concurrent and real-time systems. Its simulator can automatically visualize states or state sequences, such as counterexamples. Automatic state visualization could help human users comprehend states or state sequences, such as counterexamples, better to some extent. Each application or system, however, has its own characteristics and then must have a good picture that cannot be automatically generated but should be designed by human users especially if the main purpose of graphically animating state machines is to find out non-trivial properties that could be used as lemmas for theorem proving. A graphical user interface for Maude-NPA has been developed [11]. 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 main purpose.

Visualization of formal specifications have been attempted. Hoxha, et al. [6] have proposed how to visualize real-time temporal logic formulas because it is a error prone task to specify desired requirements in such logic for conventional engineers. Tikhonova, et al. [15] have proposed how to visualize formal specifications in a DSL implemented on the top of Event-B and Arcainiet al. [1] have proposed a visual notation for Abstract State Machines (ASMs). Unlike these studies, SMGA does not aim at visualizing formal specifications.

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 [3] and Cytoscape [12], have been developed. Some visualization techniques used in those tools could be used to implement some functionalities given by SMGA. Because state machines cannot be necessarily expressed as networks only, however, those network visualization tools cannot be directly used to graphically animate state machines.

11. Conclusion

We have described graphical animations of the Suzuki-Kasami protocol with a revised version of 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. We have summarized tips on how to design good state pictures for mutual exclusion protocols.

One piece of our future work is to graphically animate state machines that formalize other protocols than mutual exclusion protocols with SMGA or a further revised version of SMGA, although Alternating Bit Protocol (ABP) has been tackled with SMGA [8]. Another piece of our future work is to formally verify that the Suzuki-Kasami protocol enjoys the three properties by theorem proving as were done in [2]. The current implementation of SMGA is completely independent from Maude, which is pros and cons. The pros is that SMGA can play graphical animations of state sequences from another tool, such as Java Pathfinder, provided that state sequence formats conforms to SMGA. The cons is that state sequences needs to be generated in advance. It would be preferable to integrate SMGA with Maude so that state sequences generated by Maude can be visualized on-the-fly, which is yet another future work of ours.

Acknowledgment

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

References

[1] Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., 2016. Visual notation and patterns for abstract state machines, in: STAF 2016 Collocated Workshops, Revised Selected Papers, Springer. pp. 163–178. doi:10.1007/978-3-319-50230-4_12.

[2] Aung, M.T., Nguyen, T.T.T., Ogata, K., 2018. Guessing, model checking and theorem proving of state machine properties – a case study on Qlock. Intl J. Softw. Eng. & Comput. Sys. 4, 1–18. doi:doi.org/10.15282/ijsecs.4.2.2018.1.0045.

[3] Bastian, M., Heymann, S., Jacomy, M., 2009. Gephi: An open source software for exploring and manipulating networks, in: 3rd AAAI ICWSM, pp. 361–362.

[4] Bui, D.D., Ogata, K., 2019. Graphical animations of the suzuki-kasami distributed mutual exclusion protocol, in: DMSVIVA 2019, KSI Research Inc.. pp. 125–134. doi:10.18293/DMSVIVA2019-012.

[5] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All About Maude. volume 4350 of *LNCS*. Springer.

[6] Hoxha, B., Bach, H., Abbas, H., Dokhanchi, A., Kobayashi, Y., 2014. Towards formal specification visualization for testing and monitoring of cyber-physical systems, in: DIFTS14.

[7] Jackson, D., 2012. Software Abstraction. The MIT Press.

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

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

[10] Ogata, K., Futatsugi, K., 2007. Comparison of Maude and SAL by conducting case studies model checking a distributed algorithm. IEICE Trans. 90-A, 1690–1703. doi:10.1093/ietfec/e90-a.8.1690.

[11] Santiago, S., Talcott, C.L., Escobar, S., Meadows, C.A., Meseguer, J., 2009. A graphical user interface for Maude-NPA, in: 9th Spanish Conf. Prog. & Lang., pp. 3–20. doi:10.1016/j.entcs.2009.12.002.

[12] Shannon, P., Markiel, A., Ozier, O., Baliga, N.S., Wang, J.T., Ramage, D., Amin, N., Schwikowski, B., Ideker, T., 2003. Cytoscape: A software environment for integratedmodels of biomolecular interaction networks. Genome Res. 13, 2498–2504. doi:10.1101/gr.1239303.

[13] Sun, J., Liu, Y., Dong, J.S., Pang, J., 2009. PAT: Towards flexible verification under fairness, in: 21st CAV, Springer. pp. 709–714.

[14] Suzuki, I., Kasami, T., 1985. A distributed mutual exclusion algorithm. ACM TOCS 3, 344–349. doi:10.1145/6110.214406.

[15] Tikhonova, U., Manders, M., Boudewijns, R., 2016. Visualization of formal specifications for understanding and debugging an industrial DSL, in: STAF 2016 Collocated Workshops, Revised Selected Papers, Springer. pp. 179–195. doi:10.1007/978-3-319-50230-4_13.