Graphically perceiving characteristics of the MCS lock and model checking them*

Tam Thi Thanh Nguyen and Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{tamnguyen,ogata}@jaist.ac.jp

Abstract. The MCS list-based queuing lock (MCS) is a mutual exclusion protocol whose variants have been used in Java virtual machines. MCS is specified as a state machine in Maude, a rewriting logic-based computer language and system. We have developed a tool (called SMGA) that tales a finite computation generated from a state machine and displays its graphical animations. MCS is used to describe how such graphical animations help human beings perceive characteristics of the state machine of MCS. Such characteristics can be confirmed by Maude model checking facilities. The characteristics graphically perceived and confirmed by model checking could be used as lemmas to theorem prove that MCS enjoys some desired properties. SMGA can also display graphical animations of counterexamples presented by the Maude LTL model checker.
Keywords: graphical animation, Maude, model checking, mutual exclusion protocols, state machine

1 Introduction

State machines can be used as mathematical models of various systems and their properties can be used to formalize systems requirements. Thus, systems verification can be conducted as formal verification of state machine properties. Two major systems verification techniques are model checking and theorem proving. Model checking can be automatically conducted but cannot basically deal with infinite-state systems[1]. Theorem proving can directly deal with infinite-state systems but requires human interaction. One of the most intellectual tasks in theorem proving is conjecturing lemma

We have developed a tool [3] that takes a finite computation of a state machine and displays its graphical animation. The tool (called the state machine graphical animation tool, or the SMGA tool, or simply SMGA) mainly aims at

* This work was partially supported by JSPS KAKENHI Grant Number 26240008.

1 If you find a good abstraction that converts an infinite-state system to a finite-state one and preserves the negation of a property concerned, the infinite-state system can be formally verified by model checking the finite-state one [1, 2], although you need to prove the preservation of the negated property by the abstraction.

helping human beings perceive characteristics appearing in state machine graphical animations and conjecture lemmas that could be used to theorem prove state machine properties. The MCS list-based queuing lock (the MCS protocol, the MCS lock, or simply MCS) [4] is a mutual exclusion protocol whose variants have been used in Java virtual machines. MCS is specified as a state machine in Maude [5], a rewriting logic-based computer language equipped with model checking facilities (the search command and the LTL model checker). SMGA takes finite computations from the Maude specification of MCS and displays their graphical animations. This paper describes how such graphical animations help human beings perceive characteristics of the state machine of MCS appearing in the animations. The Maude search command can be used to confirm the guessed characteristics by exhaustively traversing the Maude specification of MCS. If Maude refutes some, we can revise them based on the counterexamples generated by Maude. Characteristics perceived by human beings in graphical animations and confirmed by model checking would be likely to be able to be used as lemmas for theorem proving. The paper also describes model checking experiments that MCS enjoys the mutual exclusion property with the Maude search command and the lockout freedom property with the Maude LTL model checker. Two variants of MCS in which a complex atomic instruction comp&swap is not used are analyzed with the LTL model checker as well. One variant does not enjoy the lockout freedom property and then a counterexample is given by the model checker. SMGA can also generate a graphical animation of a counterexample.

The rest of the paper is organized as follows. Sect. 2 describes some preliminaries, such as Kripke structures and LTL. Sect. 3 describes MCS. Sect. 4 describes Maude and how specify MCS in Maude. Sect. 5 reports on the case study in which MCS and two variants have been analyzed with SMGA and Maude. Sect 6 mentions some existing related work, and Sect 7 finally concludes the paper.

2 Preliminaries

Let S be a set and π be an infinite sequence $e_0; ...; e_i; \ldots$ of S, where each $e_i \in S$, and then $\pi(i) = e_i$ (the ith element in π) and $\pi^i = e_i; \ldots$ (the ith suffix obtained by deleting the first i elements from π) for each natural number i. Let $e_0; ...; e_n$ be a non-empty finite sequence of S, and then $(e_0; ...; e_n)^\infty = e_0; ...; e_n; e_0; ...; e_n; \ldots$ (the infinite sequence in which the finite sequence repeats infinitely often). Let U be a universal set of symbols.

A Kripke structure (KS) K is a 5 tuple $\langle S, I, P, L, T \rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, $P \subseteq U$ is a set of atomic state propositions, L is a labeling function whose type is $S \to 2^P$, and $T \subseteq S \times S$ is a total binary relation. An element $(s, s') \in T$ may be written as $s \to s'$ and referred as a state transition.

A path of K is an infinite sequence $s_0; \ldots; s_i; s_{i+1}; \ldots$ of S such that $(s_i, s_{i+1}) \in T$ for each natural number i. A computation of K is a path π of K such that $\pi(0) \in I$. Let $\mathcal{P}$ be the set of all paths of K and $\mathcal{C}$ be the set of all computations of K. A finite prefix $s_0; \ldots ; s_n$ of a computation (or path) of K is called a finite computation (or path) of K. The syntax of a formula φ in Linear Temporal Logic (LTL) for K is $\varphi ::= \top \mid p \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi$, where $p \in P$. Let $\mathcal{F}$ be the set of all formulas in LTL for K.

An arbitrary path $\pi \in \mathcal{P}$ of K and an arbitrary LTL formula $\varphi \in \mathcal{F}$ of K, $K, \pi \models \varphi$ is inductively defined as $K, \pi \models \top$, $K, \pi \models p$ if and only if $p \in L(\pi(0))$, $K, \pi \models \neg\varphi_1$ if and only if $K, \pi \not\models \varphi_1$, $K, \pi \models \varphi_1 \wedge \varphi_2$ if and only if $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$, $K, \pi \models \bigcirc \varphi_1$ if and only if $K, \pi^1 \models \varphi_1$, and $K, \pi \models \varphi_1 \,\mathcal{U}\, \varphi_2$ if and only if there exists a natural number i such that $K, \pi^i \models \varphi_2$ and for all natural numbers $j < i$, $K, \pi^j \models \varphi_1$, where φ_1 and φ_2 are LTL formulas. Then, $K \models \varphi$ if and only if $K, \pi \models \varphi$ for each computation $\pi \in \mathcal{C}$ of K.

The temporal connectives $\bigcirc$ and $\mathcal{U}$ are called the next operator and the until operator, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond \varphi \triangleq \top \,\mathcal{U}\, \varphi$, $\Box \varphi \triangleq \neg(\Diamond \neg\varphi)$, and $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond \varphi_2)$. The temporal connectives $\Diamond$, $\Box$ and $\rightsquigarrow$ are called the eventually operator, the always operator and the leadsto operator, respectively.

There are multiple possible ways to express states. In this paper, a state is expressed as an associative-commutative collection of name-value pairs, where a name may have parameters. Associative-commutative collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a 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. Since the order is irrelevant because of associativity and commutativity, $oc_1\ oc_2\ oc_3$ is the same as some others, such as $oc_3\ oc_2\ oc_1$. For soups ocs_1, ocs_2 of observable components, $ocs_1 \subseteq ocs_2$ if and only if there exists a soup ocs_3 of observable components such that $ocs_1\ ocs_3 = ocs_2$, namely that there exists ocs_1 in ocs_2, where each ocs_i for $i = 1, 2, 3$ may be empty or a single observable component. Examples of observable components are `(glock: nop)` and `(pc[p1]: rs)`, where `glock` and `pc[p1]` are names, `nop` and `rs` are values, and `p1` is a parameter of the name `pc[p1]`. An example of a soup of observable components is `(glock: nop) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)`. This represents (actually partially) a state in which there are three processes each of which is located at rs and there is one global variable *glock* that is shared by the three processes and whose value is nop. Since the soup is associative and commutative, even if some observable components are swapped, for example `(pc[p2]: rs) (pc[p1]: rs) (glock: nop) (pc[p3]: rs)`, it represents the same state.

3 MCS List-based Queuing Lock

The MCS list-based Queuing lock (MCS protocol) has been invented by John M. Mellor-Crummey and Michael L. Scott [4]. Variants of MCS protocol have been used in Java virtual machines, and therefore the inventors were awarded the 2006 Edsger W. Dijkstra Prize in Distributed Computing[2].

A pseudo-code of MCS protocol for each process p is as follows:

```
rs:  "Remainder Section"
l1:  $next_p$ := nop;
l2:  $pred_p$ := fetch&store($glock, p$);
l3:  if $pred_p \neq$ nop {
l4:    $lock_p$ := true;
l5:    $next_{pred_p}$ := $p$;
l6:    repeat while $lock_p$; }
cs:  "Critical Section"
l7:  if $next_p$ = nop {
l8:    if comp&swap($glock, p$, nop)
l9:      goto rs;
l10:   repeat while $next_p$ = nop; }
l11: $locked_{next_p}$ := false;
l12: goto rs;
```

There is one global variable *glock* shared by all processes participating in MCS protocol. Its type is process IDs (or Pid). Initially, *glock* is nop, a dummy process ID. Each process p maintains three local variables $next_p$, $lock_p$ and $pred_p$ whose types are Pid, Bool and Pid, respectively. Initially, $next_p$, $lock_p$ and $pred_p$ are nop, false and nop, respectively. $next_p$ is used to construct a global queue of processes (or process IDs). Basically, $next_p$ refers to the next element of the queue if p is in the queue. Since enqueuing an element into the queue and dequeuing the queue are not atomically done, however, $next_p$ may be nop even though p is not the bottom element of the queue. $pred_p$ refers to the previous element of the queue while p is being put into the queue. $lock_p$ is the local lock on which process p is spinning while $lock_p$ is true to wait for entering the critical section. *glock* basically refers to the bottom element if the queue is not empty. Since the two basic operations to the queue are not atomic, however, *glock* may not refer to the real bottom element while some process IDs are being put into the queue.

To safely conduct the two basic operations to the queue non-atomically, two atomic operations are used: fetch&store and comp&swap. fetch&store(x, v) does the following atomically: $tmp := x$, $x := v$, and tmp is returned, where tmp is a temporary variable. comp&swap(x, v_1, v_2) does the following atomically: if $x = v_1$, then $x := v_2$ and true is returned; otherwise, false is returned.

4 Maude

Maude is a rewriting logic-based computer language and system that is equipped with many functionalities, among which are model checking and meta-programming. Maude is one of the direct successors of OBJ3, the most famous algebraic specification language and system mainly designed by Joseph A. Goguen.

[2] https://www.podc.org/dijkstra/2006-dijkstra-prize/

Therefore, Maude allows users to write specifications very flexibly. For example, associative and/or commutative binary operators can be freely used in specifications, making it possible to specify complex concurrent and distributed systems very succinctly.

As described, MCS protocol is formalized as a state machine whose states are expressed as soups of observable components. When there are three processes, a state is expressed as

```
(glock: G) (pc[p1]: L1) (pc[p2]: L2) (pc[p3]: L3) (next[p1]: P1)
(next[p2]: P2) (next[p3]: P3)(lock[p1]: B1) (lock[p2]: B2) (lock[p3]: B3)
(pred[p1]: Q1) (pred[p2]: Q2) (pred[p3]: Q3)
```

where `G`, `P`i and `Q`i for $i = 1, 2, 3$ are process IDs, `L`i for $i = 1, 2, 3$ are locations, such as rs, l1 and cs, and `B`i for $i = 1, 2, 3$ are Booleans. Initially, `G`, each `P`i and each `Q`i are nop, each `L`i is rs, each `B`i is false. The initial state will be referred as `init`.

The state transitions are described in terms of rewrite rules as follows:

```
rl [want] : (pc[P]: rs) => (pc[P]: l1) .
rl [stnxt] : (pc[P]: l1) (next[P]: Q) => (pc[P]: l2) (next[P]: nop) .
rl [stprd] : (glock: Q) (pc[P]: l2) (pred[P]: Q1)
    => (glock: P) (pc[P]: l3) (pred[P]: Q) .
rl [chprd] : (pc[P]: l3) (pred[P]: Q)
    => (pc[P]: (if Q == nop then cs else l4 fi)) (pred[P]: Q) .
rl [stlck] : (pc[P]: l4) (lock[P]: B) => (pc[P]: l5) (lock[P]: true) .
rl [stnpr] : (pc[P]: l5) (pred[P]: Q) (next[Q]: Q1)
         => (pc[P]: l6) (pred[P]: Q) (next[Q]: P) .
rl [chlck] : (pc[P]: l6) (lock[P]: false) => (pc[P]: cs) (lock[P]: false) .
rl [exit] : (pc[P]: cs) => (pc[P]: l7) .
rl [rpnxt] : (pc[P]: l7) (next[P]: Q) => (pc[P]: (if Q == nop then l8
                 else l11 fi)) (next[P]: Q) .
rl [chglk] : (glock: Q) (pc[P]: l8) => (glock: (if Q == P then nop
            else Q fi)) (pc[P]: (if Q == P then l9 else l10 fi)) .
rl [go2rs] : (pc[P]: l9) => (pc[P]: rs) .
crl [rpnxt2] : (pc[P]: l10) (next[P]: Q) => (pc[P]: l11)
              (next[P]: Q) if Q =/= nop .
rl [stlnx] : (pc[P]: l11) (next[P]: Q) (lock[Q]: B)
     => (pc[P]: l12) (next[P]: Q) (lock[Q]: false) .
rl [gotrs] : (pc[P]: l12) => (pc[P]: rs) .
```

where `want`, `stnxt`, etc. are the labels of the rewrite rules.

5 Analysis of MCS Protocol

5.1 Invariant Model Checking with Search

For a state machine specification in Maude, a state `S`, a pattern `P` and a condition `C`, the Maude search command exhaustively traverses the reachable states from `S` to find states that match `P` and satisfy `C`:

```
search [N] in Mod : S =>* P such that C .
```

where `N` is a natural number. The search command tries to find at most N solutions. Note that a solution is basically a state `A` that matches `P` and satisfies `C`, but since there may be more than one substitution σ such that $\sigma(\texttt{P}) = \texttt{A}$, there may be more solutions than the number of such states and such substitutions are called solutions of the search.

The mutual exclusion property that should be enjoyed by mutual exclusion protocols, such as MCS protocol, says that there exists at most one process in the critical section at any given moment. Therefore, the search command can be used to check if MCS protocol enjoys the property as follows:

```
search [1] in MCS-INIT :
init =>* (pc[I]: cs) (pc[J]: cs) S .
```

where `MCS-INIT` is the module in which MCS protocol is specified in Maude, `I` and `J` are Maude variables of process IDs, and `S` is a Maude variable of states (or soups of observable components). If Maude finds a solution, MCS protocol does not enjoy the property. Maude did not find any solutions, implying that MCS protocol enjoys the property when there are three processes.

5.2 Graphical Animations of MCS Protocol

The graphical animation tool [3] has been implemented with DRAW-SVG [6] that is designed and developed by Joseph LIARD. It is a free online drawing application for designers and developers, making it possible to create fully standard compliant SVG. We have used DRAW-SVG as an integrated drawing tool within our tool to support users draw SVG pictures for any state machines. Our tool is available on the website `https://tamntt.bitbucket.io/Research/GraphicalAnimation/`. It allows users to design their own pictures of animations. Fig. 1 shows the picture we have drawn for MCS protocol when there are three processes.

The graphical animation tool does not deal with state machines themselves internally. Instead, what is fed into the tool is basically a finite computation of a state machine. An example input file of M_{MCS} is as follows:

```
###keys
glock pc[p1] pc[p2] pc[p3] next[p1] next[p2] next[p3] lock[p1] lock[p2]
lock[p3] pred[p1] pred[p2] pred[p3]

###textDisplay

###states
(glock: nop (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop) ||
(glock: nop (pc[p1]: rs) (pc[p2]: l1) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
```

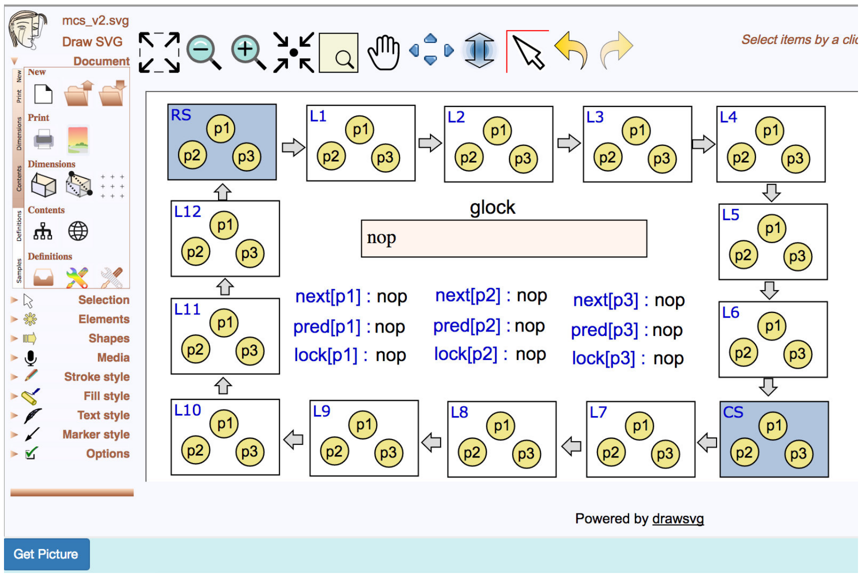


Fig. 1. Picture of MCS Protocol

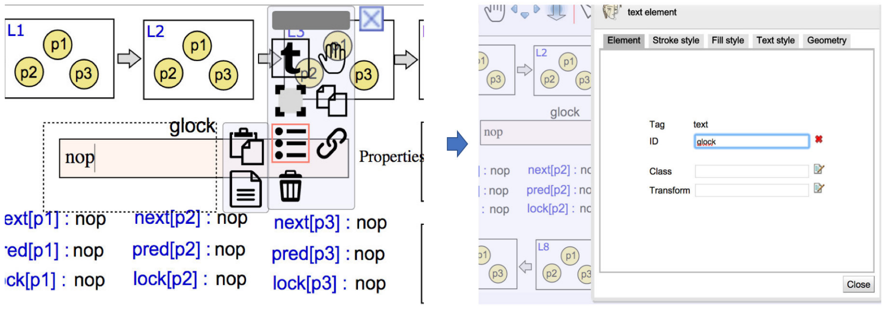
```
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop) ||
glock: nop (pc[p1]: l1) (pc[p2]: l1) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop
```

There are three segments in an input file as follows:

– ###*keys*: This is a list of keys which are names of observable components in a state. The order in which the keys appear must be the same as the order in which the corresponding observable components appear in each state.
– ###*textDisplay*: This part specifies how the value of an observable component is displayed. If nothing is specified, it is displayed horizontally and its top appears left most when displaying a queue or string list. There may be the case, however, where its top should appear right most. Some values, such as stacks, may have to be displayed vertically instead. The format used in this part is as follows:

  ```
  key:::option:::regex(0)++++....++++regex(i)
  ```

  The format consists of three parts: key, option and regexs. A key appearing in the key segment is written in the key part. REV, VER or VER-REV is written in the option part. REV specifies a collection, such as queues and lists, is displayed such that its top appears right most, VER specifies a collection, such as stacks, is displayed vertically such that its top appears top most, and VER-REV specifies a collection is displayed vertically such that its top appears bottom most. A list of regular expressions is written in



Tag text
ID glock

Fig. 2. Setting the property ID for displaying values of *glock* of M_{MCS}

the regexs part. For example, we have an observable component in a state as $(chan1 :< false, pac(1) >< true, pac(2) > empty)$, and we want the tool will display value of chan1 as $empty < true, pac(2) >< false, pac(1) >$, the textDisplay segment is as follows:

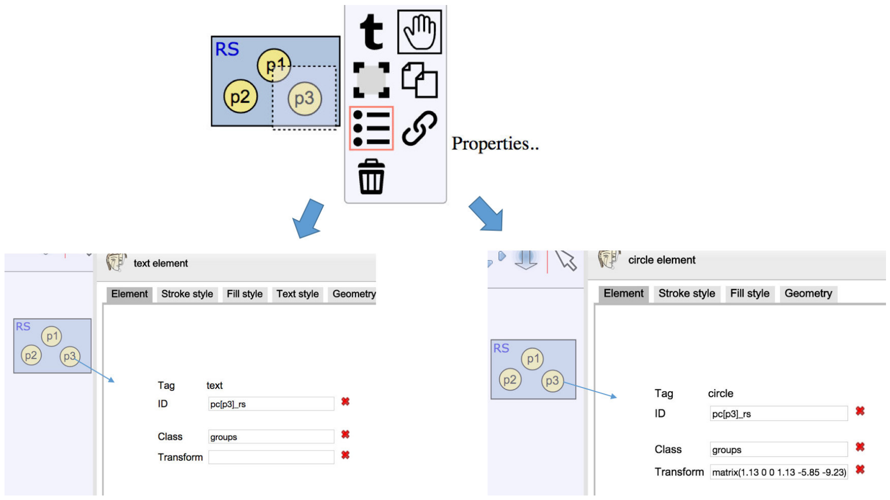
```
chan1::::REV::::<_,_>++++empty
```

Two regular expressions `<_,_>` and `empty` are written in the regexs part. They match texts, such as `<false,p(1)>`, `<true,p(2)>`, and `empty`. For the case MCS, nothing is specified in the `###textDisplay` part since values of observable components are displayed horizontally and theirs top appear left most.

– *###states*: This is a finite computation of a state machine, namely a finite sequence of states. The sign || is a separator used to distinguish adjacent states.

After drawing the picture of a state machine, the user needs to edit properties for texts on the picture so that the observable components of the state machine can appear on the picture when the state machine is animated. As clicking a text on the picture and choosing the icon of properties, a pop-up will be displayed for editing properties. In this pop-up, the *name* as an ID for the text of an observable component $(name : value)$ is set for the text so that the *value* can be displayed at the place where the text is located. The ID will be used for mapping it to the values whose name is *name* appearing in an input data when we run the graphical animation tool. For example, Fig. 2 shows *glock* is set as the ID of the observable component $(glock : nop)$ so that the *nop* is displayed at the designated place on a state machine picture.

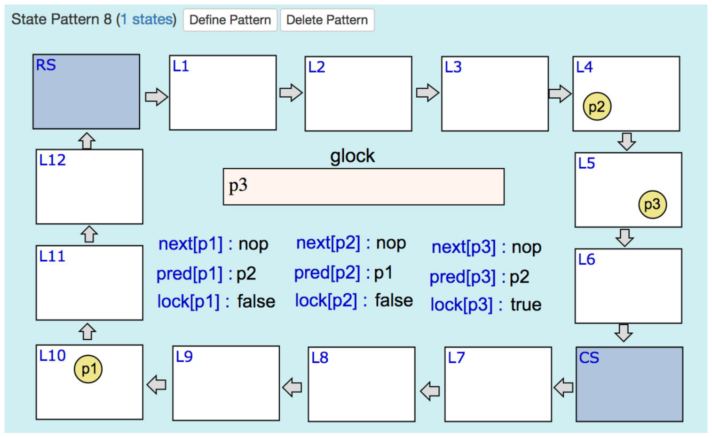
On the other hand, we want to display $(name : value)$ pairs at different locations such as process $p1$ at cs, process $p2$ at rs, ... Thus, we can draw SVG elements as rectangles to display for locations such as rs, cs, ..., and draw circles with texts for displaying every processes for every location. Then, we will set properties for the circle and the text of every process at every location. The property *class* of them will be also set is *groups*. And the property ID of the circle, and the text of every process will be set as structure KEY_VALUE, where KEY is the name, and $VALUE$ is the value of a name-value pair. By this



Tag text
ID pc[p3]_rs
Class groups
Transform

Tag circle
ID pc[p3]_rs
Class groups
Transform matrix(1.13 0 0 1.13 -5.85 -9.23)

Fig. 3. Setting properties such as *Class*, *ID* for displaying the process $p3$ at the location rs of M_{MCS}
.



glock: p3

next[p1] : nop
pred[p1] : p2
lock[p1] : false

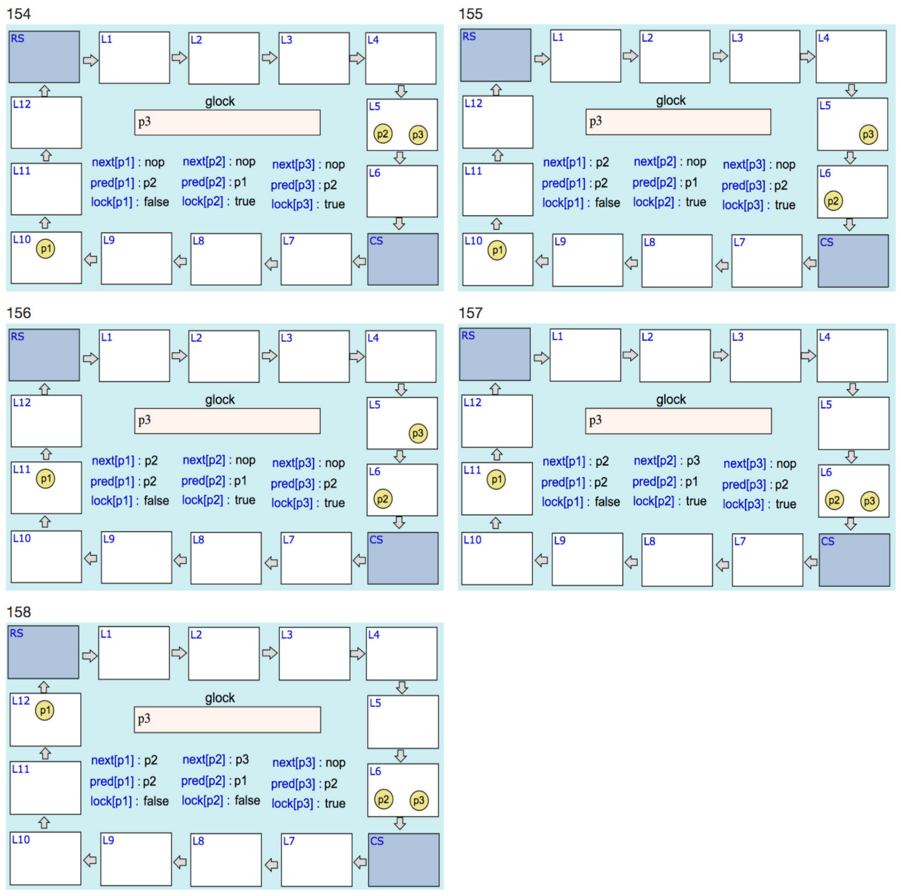
next[p2] : nop
pred[p2] : p1
lock[p2] : false

next[p3] : nop
pred[p3] : p2
lock[p3] : true

Fig. 4. A state such that `p1` is at `l10`

way, we can see that locations of processes are changed and displayed graphically when the tool animate states. For example, Fig. 3 shows how to set properties for the process $p3$ at location rs. To display the process $p3$ at the location rs, we will set the property ID is $pc[p3]\_rs$ for both the circle, and text element which visualize process $p3$. And we will also set the property $class$ is $groups$ for them.

After getting a drawn picture of a state machine and importing a prepared input file, the tool can run to play a graphical animation of the state machine. The tool allows human users to adjust the duration of the speed of animation. The unit of duration is millisecond. The smaller the duration is, the faster the



154: glock p3

next[p1] : nop   next[p2] : nop   next[p3] : nop
pred[p1] : p2   pred[p2] : p1   pred[p3] : p2
lock[p1] : false   lock[p2] : true   lock[p3] : true

155: glock p3

next[p1] : p2   next[p2] : nop   next[p3] : nop
pred[p1] : p2   pred[p2] : p1   pred[p3] : p2
lock[p1] : false   lock[p2] : true   lock[p3] : true

156: glock p3

next[p1] : p2   next[p2] : nop   next[p3] : nop
pred[p1] : p2   pred[p2] : p1   pred[p3] : p2
lock[p1] : false   lock[p2] : true   lock[p3] : true

157: glock p3

next[p1] : p2   next[p2] : p3   next[p3] : nop
pred[p1] : p2   pred[p2] : p1   pred[p3] : p2
lock[p1] : false   lock[p2] : true   lock[p3] : true

158: glock p3

next[p1] : p2   next[p2] : p3   next[p3] : nop
pred[p1] : p2   pred[p2] : p1   pred[p3] : p2
lock[p1] : false   lock[p2] : false   lock[p3] : true

Fig. 5. States 154, 155, 156, 157 and 158

animation is played. Animations can be played step by step in addition to that they can be played automatically from the beginning to the end. When an animation is played step by step, we can observe each state transition graphically.

The graphical animation tool basically takes a sequence of states and plays it graphically. The main purpose of the tool is to help human users recognize some useful patterns in animated computations, and therefore it is necessary to generate a long sequence of states. The Maude search command can generate sequences of states, but cannot generate very long sequences, such as a sequence that consists of 100 or more states due to the state explosion problem. Thus, we had written a meta-program in Maude to generate a long sequence of states. We used the meta-program to generate a finite computation that consists of 200 states for MCS protocol.

The tool can select and display the states that satisfy a condition from the input finite computation. The format of a defined condition is as follows:

```
(state['key1'] op1 state['key2']) op2 (state['key3'] op4 'value') ...
```

where `key1`, `key2`, and `key3` are names of observable components in states and keys appearing in the key segment of an input file, `op1`, `op2`, and `op3` are JavaScript comparison and logical operators, and `value` is a value. We asked the tool to select and display the states such that the location of `p1` is `l10` by using a condition that is defined as ($state$['pc[$p1$]'] =='$l10$'). The tool found 16 such states in the input finite computation. Fig. 4 shows one of the 16 states. The tool lets us know the state appear in the input finite computation at position 153. In the state, since `p1` is at l10, `p1` is dequeuing the global queue, while since `p2` and `p3` are `l4` and `l5`, `p2` and `p3` are enqueuing `p2` and `p3` into the global queue, respectively, but none of them has completed. Given a state number n, the tool displays the state at position n. We asked the tool to display the state at position 153 and play the animation from the state step by step. Fig. 5 shows the five states at positions 154, 155, 156, 157 and 158 from the top. In state 153, `p2` executes the assignment at l4, setting $lock_{\texttt{p2}}$ true, and moves to l5 but has not yet completed enqueuing `p2` into the global queue. In state 154, `p2` executes the assignment at l5, setting $next_{\texttt{p1}}$ to `p2`, and moves to l6, when `p2` has eventually completed enqueuing `p2` into the global queue. In state 155, `glock` is `p3`, meaning that `p3` is the bottom element of the global queue but `p3` has not completed enqueuing `p3` into the global queue. In state 155, `p1` leaves the loop at l10 and moves to l11 but has not yet completed dequeuing the global queue. In state 156, `p3` executes the assignment at l5, setting $next_{\texttt{p2}}$ to `p3`, and moves to l6, when `p3` has eventually completed enqueuing `p3` into the global queue. In state 157, `p1` executes the assignment at l11, setting $lock_{\texttt{p2}}$ false, letting know `p2` is ready to enter the critical section, and moves to l12. In state 158, the global queue consists of `p2` and `p3` in this order because $lock_{\texttt{p2}}$ is `false`, $next_{\texttt{p2}}$ is `p3`, $lock_{\texttt{p2}}$ is `true`, $next_{\texttt{p2}}$ is `nop`, and `glock` is `p3`.

5.3 Perceiving Characteristics with Graphical Animations

By observing graphical animations of MCS, we have also found some characteristics or patterns appearing in them. Although we do not prove those characteristics in this paper, we will model check them in the next sub-section. Proving the characteristics is one piece of our future work. In this sub-section, we present some characteristics guessed by observing graphical animations of a finite computation $FC1000$ that consists of 1000 states as follows:

$Characteristic$ 1:

If there is a process in the critical section cs, the local lock owned by each process that wants to enter the cs is true.

$Characteristic$ 2:

If a process p is at the location $l3$ and $pred[p] = nop$, there is no process in the critical section cs.

$Characteristic$ 3:

If a process p is at the location $l6$ and $lock[p] = false$, there is no process in the critical section.

$Characteristic$ 4:

> If a process $p1$ is at the location $l12$, another process $p2$ is at $l6$, and $pred[p2] = p1$ then the $lock[p2]$ is $false$.

$Characteristic$ 5:

> If a process $p1$ is at the location $l9$ and another process $p2$ is at $l4$ then the $glock$ is $p2$.

$Characteristic$ 6:

> No state such that a process is at cs, $l7$, $l8$, $l10$, or $l11$ and another state is at cs, $l7$, $l8$, $l10$, or $l11$.

$Characteristic$ 7:

> If there is a process is at $l3$, $l4$, $l5$, $l6$, cs, $l7$, $l8$, $l10$, or $l11$, $glock \neq nop$.

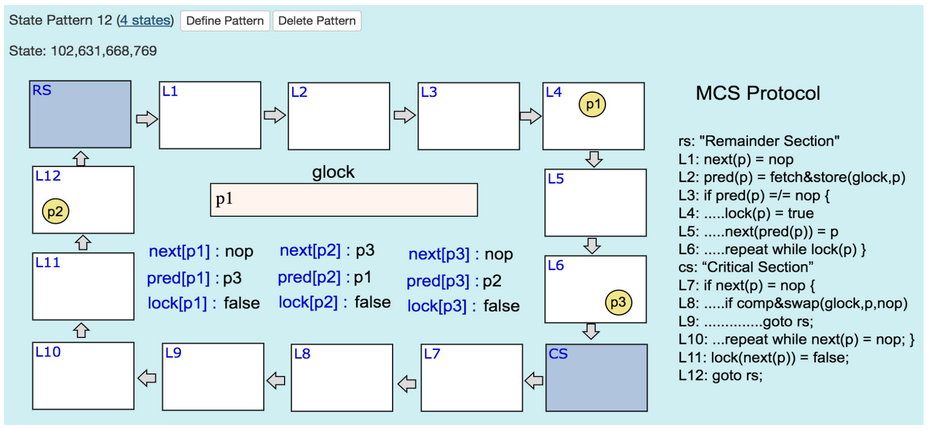
Besides taking a close look at several graphical animations of MCS to recognize some characteristics appearing in the graphical animations, we can also ask the tool to show us all states that satisfy some conditions. The tool supports us to select the states among the ones in a given input file such that a condition is fulfilled and to display their graphical representations. If some states are similar each other, they will be clustered into one state representation as a state pattern. For example, if we have a sequence of state such as A, B, C, A, D, E, B, . . . , the tool will group same states, and display different states such as A, B, C, D, E. Thus, we can define some conditions to filter satisfied states to check or confirm some predicted characteristics, and reduce the amount of time for animations observing. If the tool refutes guessed characteristics, we should correct them. An example (called $Cond1$) of the conditions is as follows:

```
((state['pc[p1]'] == 'l12' ) || (state['pc[p2]'] == 'l12' ) ||
(state['pc[p3]'] == 'l12' )) && ((state['pc[p1]'] == 'l6' )
|| (state['pc[p2]'] == 'l6' ) || (state['pc[p3]'] == 'l6' ))
```

This condition can select the states such that there is a process $p1$ in the $l12$, and there is another process $p2$ in the $l6$. By using the $Cond1$ for selecting satisfied states from an input file in which the states segment is $FC1000$, we found 130 states clustered into 66 state patterns. Fig. 6 shows a state pattern that satisfies the condition $Cond1$. This pattern is a representation of four same states such as the state 102, 631, 668, and 769. We used $Cond1$ to check a guessed characteristic (called $Pre4$) as follows:

> If there is a process $p1$ in the $l12$, and there is another process $p2$ in the $l6$ then $lock[p2] = false$.

However, we found some states do not satisfy characteristic $Pre4$. One of them is shown in Fig. 7. In this state, process $p2$ is at $l12$, two processes $p1$ and $p3$ are at $l6$, $lock[p1] = true$, $lock[p3] = false$. Thus, we reviewed the graphically displayed states that enjoyed $Cond1$. And we perceived that if a process $p1$ is at the location $l12$, another process $p2$ is at $l6$, and $pred[p2] = p1$ then the $lock[p2]$ is $false$. This is the $Characterictic4$ which we have mentioned above. To check our guess, we made another condition (called $Cond2$) as follows:



State: 102,631,668,769

glock: p1

next[p1] : nop
pred[p1] : p3
lock[p1] : false

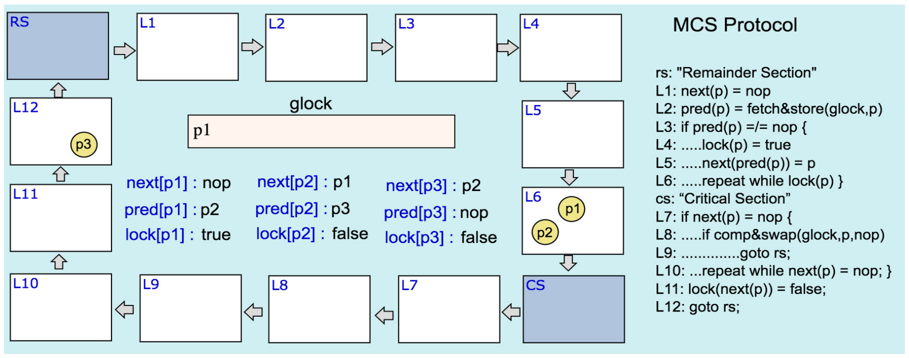
next[p2] : p3
pred[p2] : p1
lock[p2] : false

next[p3] : nop
pred[p3] : p2
lock[p3] : false

MCS Protocol

```
rs: "Remainder Section"
L1: next(p) = nop
L2: pred(p) = fetch&store(glock,p)
L3: if pred(p) =/= nop {
L4: .....lock(p) = true
L5: .....next(pred(p)) = p
L6: .....repeat while lock(p) }
cs: "Critical Section"
L7: if next(p) = nop {
L8: .....if comp&swap(glock,p,nop)
L9: .............goto rs;
L10: ...repeat while next(p) = nop; }
L11: lock(next(p)) = false;
L12: goto rs;
```

Fig. 6. A state pattern clustered from four same states, which satisfies the condition *Cond*1.



glock: p1

next[p1] : nop
pred[p1] : p2
lock[p1] : true

next[p2] : p1
pred[p2] : p3
lock[p2] : false

next[p3] : p2
pred[p3] : nop
lock[p3] : false

MCS Protocol

```
rs: "Remainder Section"
L1: next(p) = nop
L2: pred(p) = fetch&store(glock,p)
L3: if pred(p) =/= nop {
L4: .....lock(p) = true
L5: .....next(pred(p)) = p
L6: .....repeat while lock(p) }
cs: "Critical Section"
L7: if next(p) = nop {
L8: .....if comp&swap(glock,p,nop)
L9: .............goto rs;
L10: ...repeat while next(p) = nop; }
L11: lock(next(p)) = false;
L12: goto rs;
```

Fig. 7. A state does not satisfy the characteristic *Pre*4.

```
((state['pc[p1]'] == 'l12') && (((state['pc[p2]'] == 'l6') &&
(state['pred[p2]'] == 'p1')) || ((state['pc[p3]'] == 'l6') &&
(state['pred[p3]'] == 'p1')))) || ((state['pc[p2]'] == 'l12')
&& (((state['pc[p1]'] == 'l6') && (state['pred[p1]'] == 'p2'))
|| ((state['pc[p3]'] == 'l6') && (state['pred[p3]'] == 'p2'))))
|| ((state['pc[p3]'] == 'l12') && (((state['pc[p2]'] == 'l6')
&& (state['pred[p2]'] == 'p3')) || ((state['pc[p1]'] == 'l6')
&& (state['pred[p1]'] == 'p3'))))
```

This condition can select the states such that there is a process $p1$ in the $l12$, and there is another process $p2$ in the $l6$ such that $pred[p2] = p1$. We found 111 states clustered into 51 state patterns, and perceived that all of them satisfied $Characterictic4$.

Thus, the tool supports us usefully to perceive some characteristics appearing graphical animations. When paired with an intuitively designed picture and sequence of states, the tool can create well-crafted animated visualizations which are effective at attracting viewers and supporting them to more conveniently understand complex characteristics. After using graphical animations to get better understandings of M_{MCS} and recognizing some characteristics, we will model check the characteristics by Maude to confirm them. Some of them may be refuted by model checking, which allows human beings to revise such characteristics based on the counterexamples given by a model checker and may help them get better understandings of M_{MCS}.

5.4 Confirming Characteristics with Maude

In this sub-section, we describe how to confirm the characteristics with the search command in Maude.

We asked the Maude search command to find a state such that a given characteristic (or property) is broken. If Maude finds a solution, the characteristic is not enjoyed by *MCS* protocol. Otherwise, MCS enjoys the characteristic (or the property) when there are three processes because we use them in the model checking experiments. The seven characteristics guessed with the help of the state machine graphical animation tool can be confirmed with the Maude search command:

– Characteristic 1:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (lock[J]: B)
S such that not ((L1 == cs and (L2 == l6)) implies B) .
```

– Characteristic 2:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pred[I]: K) (pc[J]: L2)
S such that not ((L1 == l3 and K == nop) implies not (L2 == cs)) .
```

– Characteristic 3:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (lock[I]: B) (pc[J]: L2)
S such that not ((L1 == l6 and not B) implies not (L2 == cs)) .
```

– Characteristic 4:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (lock[J]: B)
(pred[J]: K) S such that not ((L1 == l12 and L2 == l6 and K == I)
implies not B) .
```

– Characteristic 5:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (glock: K)
S such that not (((L1 == l9) and (L2 == l4)) implies ( K == J)) .
```

– Characteristic 6:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) S such that
(L1 == cs or L1 == l7 or L1 == l8 or L1 == l9 or L1 == l10 or
L1 == l11) and (L2 == cs or L2 == l7 or L2 == l8 or L1 == l9 or
L2 == l10 or L2 == l11) .
```

– Characteristic 7:

```
search [1] in MCS-INIT : init =>* (glock: K) (pc[I]: L) S such that
not ((L == l3 or L == l4 or L == l5 or L == l6 or L == cs or L == l7
or L == l8 or L == l10 or L == l11) implies (not K == nop)) .
```

where `MCS-INIT` is the module in which MCS protocol is specified in Maude, `I`, `J`, and `K` are Maude variables of process IDs, `L`, `L1`, and `L2` are Maude variables of locations, `B` is a Maude variable of Boolean, and `S` is a Maude variable of states (or soups of observable components). Each of the seven search commands found no solution, meaning that MCS enjoys the seven characteristics (or the seven properties) when there are three processes. The model checking experiments, however, do not guarantee that MCS enjoys the seven characteristics for an arbitrary number of processes. We will theorem prove the seven characteristics for an arbitrary number of processes in future by writing what are called proof scores in CafeOBJ, a sibling language of Maude. We predict that some of the seven characteristics could be used as lemmas when we theorem prove that MCS enjoys the mutual exclusion property for an arbitrary number of processes.

5.5 LTL Model Checking

In this sub-section and the following sub-sections, we suppose that there are two processes `p1` and `p2` and let `init` denote the initial state in which the two processes participate in MCS protocol.

To use Maude LTL model checker, users are supposed to specify atomic propositions. Let us suppose we model check MCS protocol enjoys the lockout freedom property when there are two processes. The lockout freedom property says whenever each process wants to enter the critical section, it will eventually be there. To express the property in LTL, we need two kinds of atomic propositions `want(P)` and `crit(P)`, where `P` is a process ID. Users are also supposed to specify a labeling function. For our purpose, we declare the three equations: `eq (pc[P] : l1) S |= want(P) = true .`, `eq (pc[P] : cs) S |= crit(P) = true .`, and `eq S |= PROP = false [owise] .`, where `P` is a Maude variable of process IDs, `S` is a Maude variable of states (or soups of observable components), and `PROP` is a Maude variable of atomic propositions. The three equations say a state s satisfies `want(P)` if and only if `(pc[P]: l1)` $\subseteq s$ and s satisfies `crit(P)` if and only if `(pc[P]: cs)` $\subseteq s$.

Then, users are supposed to specify LTL formulas to check. The lockout freedom property is expressed as `wait(P)` $\leadsto$ `crit(P)`. In Maude, the formula is specified as `eq lofree(P) = (want(P) |-> crit(P)) .`, where the operator `_|->_` denotes the leadsto operator $\leadsto$.

Model checking that the Kripke structure formalizing MCS protocol satisfies the lockout freedom property `lofree(p1)` for `p1` is conducted by reducing the term `modelCheck(init,lofree(p1))`. Maude model checker generates a counterexample. This is because since we do not use any fairness assumptions, there may be cases in which only `p2` is given a processing resource, which may happen if we use some unfair scheduler. Let us suppose we use a fair scheduler. If we adopt a fair scheduler, the fair scheduler guarantees that each process terminates any non-loop statements, such as assignments. For example, if `p1` is at l1, `p1` will eventually move to either l6 or cs. To express the situation in which we adopt a fair scheduler, we use four more kinds of atomic propositions `dose(P)`, `spin1(P)`, `spin2(P)` and `exit(P)`. For them, we declare four more equations

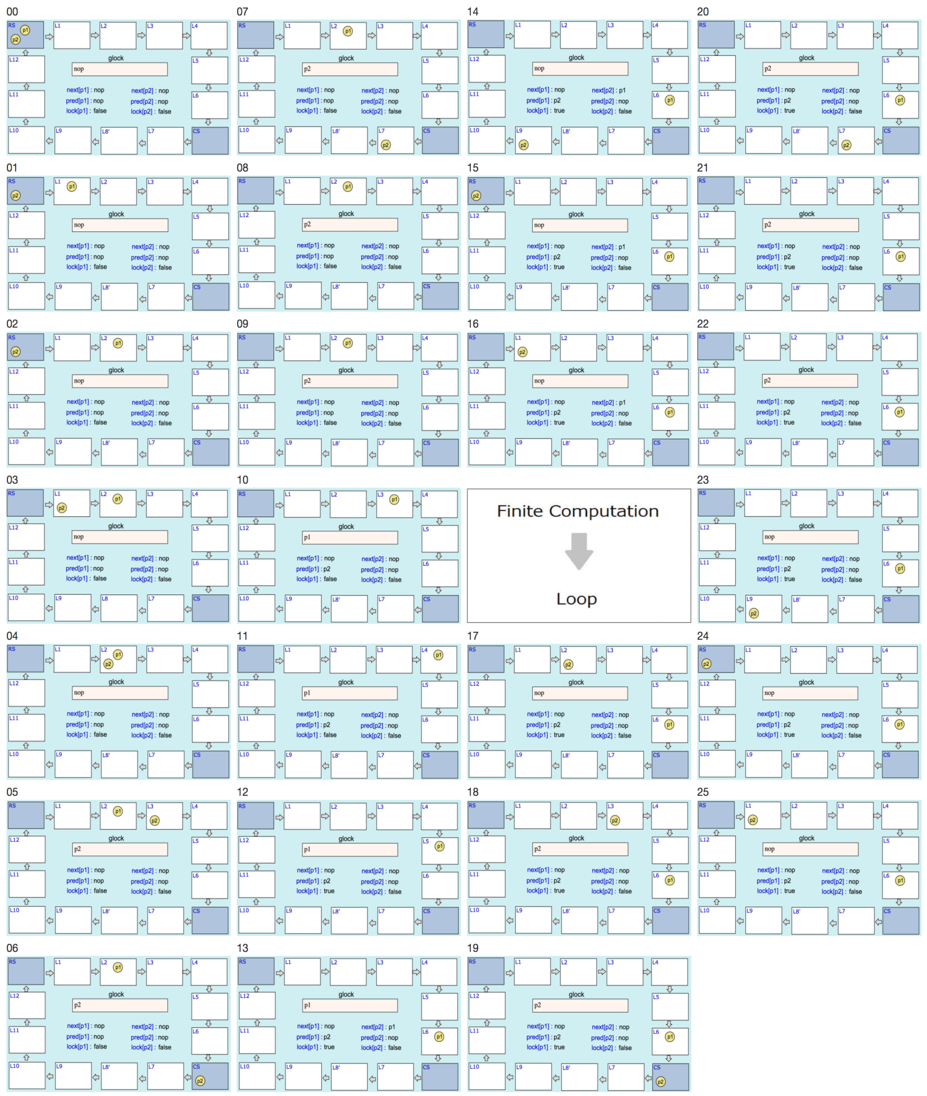
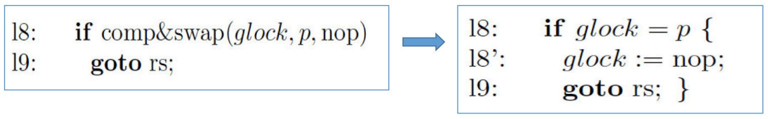


Fig. 8. A counterexample for the lockout freedom property for MCS protocol in which comp&swap is not naively used.

`eq (pc[P] : rs) S |= dose(P) = true .`, `eq (pc[P] : l6) S |= spin1(P) = true .`, `eq (pc[P] : l10) S |= spin2(P) = true .` and `eq (pc[P] : l11) S |= exit(P) = true .` in addition to the three equations for atomic propositions. The assumption used is expressed as `(want(P) |-> (crit(P) \/ spin1(P))) /\ (crit(P) |-> (dose(P) \/ spin2(P))) /\ (exit(P) |-> dose(P))` for process `P`, which will be referred as `fair(P)`. The LTL formula says that if `P` is at l2, `P` will eventually move to l6 or cs, if `P` is at cs, `P` will eventually move to l10 or



```
l8:    if comp&swap(glock, p, nop)
l9:      goto rs;
```

```
l8:    if glock = p {
l8':     glock := nop;
l9:      goto rs; }
```

Fig. 9. A modification such that `comp&swap` is disused.

rs, and if `P` is at l11, `P` will eventually move to rs. The LTL formula to express the property under the assumption is `(fair(p1) /\ fair(p2)) -> lofree(P)`, which will be referred as `lofuf(P)`, where `->` is the logical implication. Reducing `modelCheck(init,lofuf(p1))` does not generate any counterexamples, meaning that MCS protocol enjoys the lockout freedom property under the assumption that a fair scheduler is adopted when there are two processes.

The rewrite rules `chlck` and `rpnxt2` may be declared as follows:

```
rl [chlck'] : (pc[P]: l6) (lock[P]: B) => (pc[P]: (if B then l6
                                          else cs fi)) (lock[P]: B) .
rl [rpnxt2'] : (pc[P]: l10) (next[P]: Q) => (pc[P]: (if Q == nop then l10
                                          else l11 fi)) (next[P]: Q) .
```

If we use `chlck'` and `rpnxt2'` instead of `chlck` and `rpnxt2`, reducing `modelCheck(init,lofuf(p1))` generates a counterexample. This is because the assumption used does not prohibit a process only repeats a loop forever. Each of the loops used in MCS protocol does not change anything if its condition is true. Therefore, the two loops can be formalized as the rewrite rules `chlck` and `rpnxt2`, which could make the assumption simpler.

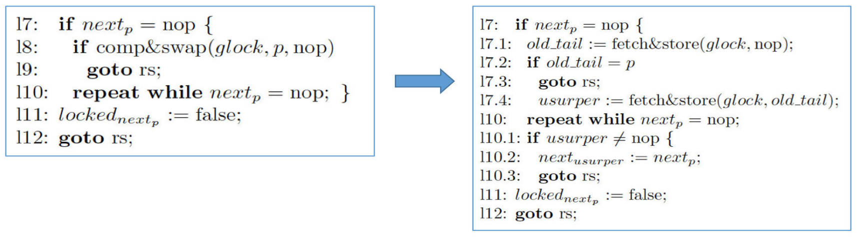
5.6 A Naive Way to Disuse `comp&swap`

MCS protocol uses two atomic operators `fetch&store` and `com&swap`. We model check the two properties for a variant of MCS protocol in which `comp&swap` is not used. Fig. 9 shows how to change the protocol.
The rewrite rule `chglk` is then replaced with the following two rewrite rules:

```
rl [chglk'] : (glock: Q) (pc[P]: l8) => (glock: Q) (pc[P]: (if Q == P
                                            then l8' else l10 fi)) .
rl [stglk] : (glock: Q) (pc[P]: l8') => (glock: nop) (pc[P]: l9) .
```

Model checking the two properties for the variant, the search command does not find any counterexamples for the mutual exclusion property but the LTL model checker finds a counterexample for the lockout freedom property even if a fair scheduler is adopted. Note that we can use exactly the same assumption used to model check that MCS protocol enjoys the lockout freedom property.

A counterexample generated by Maude LTL model checker consists of a finite computation from an initial state to a state leading to an infinite loop such that a finite state sequence is repeated forever or leading to a deadlock state. The counterexample generated by Maude LTL model checker for the lockout freedom under the use of a fair scheduler consists of a finite computation that consists of 17 states leading to an infinite loop such that a finite state sequence that consists



```
l7:   if $next_p$ = nop {
l8:     if comp&swap($glock$, $p$, nop)
l9:       goto rs;
l10:    repeat while $next_p$ = nop; }
l11:  $locked_{next_p}$ := false;
l12:  goto rs;
```

```
l7:    if $next_p$ = nop {
l7.1:    $old\_tail$ := fetch&store($glock$, nop);
l7.2:    if $old\_tail = p$
l7.3:      goto rs;
l7.4:      $usurper$ := fetch&store($glock$, $old\_tail$);
l10:     repeat while $next_p$ = nop;
l10.1: if $usurper \neq$ nop {
l10.2:   $next_{usurper} := next_p$;
l10.3:   goto rs;
l11:   $locked_{next_p}$ := false;
l12:   goto rs;
```

Fig. 10. A correction of the wrong part.

of 9 states is repeated forever. We had extended the state machine graphical animation tool so that a counterexample can be graphically animated [7]. Feeding the counterexample generated by Maude LTL model checker, the extended tool graphically animates it, repeating the loop part, which lets us realize only `p2` enters and leaves the critical section repeatedly while `p1` is waiting at l6 until $lock_{\texttt{p1}}$ becomes false. Fig. 8 shows the 26 pictures of the states composing the counterexample. The first 17 states is the finite computation, while the last 9 states is the finite state sequence that repeats forever, making the loop. Note that state 0 is the top state of the finite computation.

In state 8, `p1` is at l2 and is enqueuing it into the global queue, and `p2` is at l8 and is dequeuing the global queue. `p2` checks the condition of the **if** statement at l8. Since $glock$ is not `p2`, `p2` moves to l8'. In state 9, `p1` executes $pred_{\texttt{p1}}$:= fetch&store($glock$, `p1`); at l2, making $glock$ `p1` and $pred_{\texttt{p1}}$ `p2`. In state 9, since $pred_{\texttt{p1}}$ is `p2`, the predecessor of `p1` is `p2` in the global queue, meaning that `p1` has not been extracted from the global queue. In what follows, since $pred_{\texttt{p1}}$ is not nop, `p1` sets $next_{\texttt{p2}}$ to `p1` and $lock_{\texttt{p1}}$ true, and waits at l6 until $lock_{\texttt{p1}}$ becomes false. In state 13, `p2` executes $glock$:= nop; at l8'. Therefore, in state 14, $glock$ is nop, meaning that the global queue is empty, although `p1` is waiting at l6 until $lock_{\texttt{p1}}$ becomes false. This is way `p1` is waiting at l6 forever and only `p2` enters and leaves the critical section repeatedly.

5.7 The Mellor-Crummey & Scott's Way to Disuse comp&swap

Mellor-Crummey & Scott have also proposed how to implement MCS protocol such that `comp&swap` is not used. Fig. 10 shows their way to disuse `comp&swap`.

Accordingly, the Maude specification of MCS has been revised, and model checking for the lockout freedom property has been conducted. No counterexample was found.

6 Related Work

MCS protocol has been formally verified. Farn Wang [8] has automatically conducted formal proof that MCS protocol enjoys the mutual exclusion property for an arbitrary number of processes but has not for the lockout freedom property. Farn Wang and Karsten Schmidt [9] have proposed a way to formally conduct symmetric symbolic safety-analysis of concurrent software with pointer data structures. MCS protocol has been used as an example to demonstrate the proposed technique. They only consider the mutual exclusion property but not the lockout freedom property. The second author of the present paper and Kokichi Futatsugi [10] have semi-formally proved that MCS protocol enjoys both the mutual exclusion property and the lockout freedom property. Neither of them has taken into account the two variants of MCS protocol in which comp&swap is naively disused and the Mellor-Crummey & Scott's way to disuse of comp&swap is used. Gerhard Schellhorn, Oleg Travkin and Heike Wehrheim [11] have proposed a way to prove concurrent programs enjoy the lockout (or starvation) freedom property. They proved MCS protocol as an example enjoys the property. However, none of them has graphically animated MCS protocol.

Most formal specification languages, such as Z, B method, and Event-B, are not executable, although some, such as VDM and VDM++, are semi-executable. Therefore, some researches have been carried out, making formal specifications written in such languages run, for example, by translating sub-sets of such languages into programming languages. Running formal specifications is called specification animation. Specification animation makes it possible to help human users get better understandings of formal specifications. Therefore, specification animation have been used to improve some other activities, such as refinement [12, 13], inspection and formal specification construction [14, 15], and software monitoring [16]. Although specification animation does not necessarily mean visual and graphical animations, some tools make it possible to play graphical animations [15]. The formal specification language we have used is Maude. Since Maude is executable, we do not need to develop any translators. Our approach to use of Maude and the state machine graphical animation tool has been directing a similar goal to those of these existing studies.

The second author of the present paper and K. Futatsugi have semi-formally proved that MCS protocol enjoys both the mutual exclusion property and the lockout freedom property, but have not formally proved. The semi-formal proofs may have overlooked several subtle lemmas. The main purpose of the state machine graphical animation tool helps human users recognize some useful patterns from graphically animated computations. from which human users could conjecture useful lemmas [3]. One piece of our future work is to recognize useful patterns from several graphically animated computations, conjecture useful lemmas from the animated computations and formally verify MCS protocol enjoys the mutual exclusion property and the lockout freedom property.

7 Conclusion

MCS was used to demonstrate how graphical animations of the state machine of MCS help human beings perceive characteristics of the state machine appearing in the animations. Graphical animations of a state machine are generated by SMGA from finite computations of the state machine. Such guessed characteristics can be confirmed by the Maude search command. If a counterexample is found for a guessed characteristic, we could revise the characteristic based on the counterexample. The characteristics graphically perceived and confirmed could be used as lemmas to theorem prove that MCS enjoys desired properties, such as the mutual exclusion property and the lockout freedom property. We also described the model checking experiments with the Maude LTL model checker that MCS and two variants enjoy the mutual exclusion property.

References

1. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16** (1994) 1512–1542
2. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. **403** (2008) 239–264
3. Nguyen, T.T.T., Ogata, K.: Graphical animations of state machines. Submitted for Publication (https://tamntt.bitbucket.io/Research/Paper/smga.pdf) (2017)
4. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9** (1991) 21–65
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. LNCS 4350. Springer (2007)
6. Liard, J.: Draw SVG website. http://www.drawsvg.org/ (2015)
7. Nguyen, T.T.T., Ogata, K.: A way to comprehend counterexamples generated by Maude LTL model checker. Submitted for Publication (2017)
8. Wang, F.: Automatic verification of pointer data-structure systems for all numbers of processes. In: FM' 99 – Formal Methods. LNCS 1708, Springer (1999) 328–347
9. Wang, F., Schmidt, K.: Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In: 22nd FORTE. LNCS 2529, Springer (2002) 50–64
10. Ogata, K., Futatsugi, K.: Formal verification of the MCS list-based queuing lock. In: 5th ASIAN. LNCS 1742, Springer (1999) 281–293
11. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: 12th iFM. LNCS 9681, Springer (2016) 193–209
12. Hallerstede, S., Leuschel, M., Plagge, D.: Refinement-animation for Event-B - towards a method of validation. In: ABZ 2010. LNCS 5977, Springer (2010) 287–301
13. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. Sci. Comput. Program. **78** (2013) 272–292
14. Liu, S.: Validating formal specifications using testing-based specification animation. In: FormaliSE@ICSE 2016. (2016) 29–35
15. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. IEEE Trans. Reliability **65** (2016) 88–106
16. Liang, H., Dong, J.S., Sun, J., Wong, W.E.: Software monitoring through formal specification animation. ISSE **5** (2009) 231–241