Formal specification and model checking of a recoverable wait-free version of MCS

Duong Dinh Tran, Kentaro Waki, and Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {duongtd,kentaro.waki,ogata}@jaist.ac.jp

***Abstract*—MCS is widely known as one of the most efficient and influential spinning lock mutual exclusion protocols. The protocol, however, only works under the assumption that processes do not crash while acquiring/releasing the lock or being in the critical section. Furthermore, the exit segment pseudo-code of MCS's algorithm is not wait-free since a process releasing the lock needs to wait for the next process in the virtual queue to perform some steps. A new version of MCS has been proposed by S. Dhoked and N. Mittal such that the new version is wait-free and recoverable (i.e., if some processes crash, the protocol can recover and work normally). In this paper, we formally specify the recoverable wait-free version of MCS and conduct model checking to check whether the protocol enjoys the mutual exclusion property. Our experiments say that: (1) the property is not satisfied if crashes are allowed to occur without any restriction, (2) the protocol enjoys the property if crashes never happen at all, or (3) if crashes have not occurred recently. We also describe the challenge of how to formally specify dynamic memory allocation and present our solution to solve that problem.**

***Keywords*-mutual exclusion; MCS protocol; wait-free algorithm; recoverable; dynamic allocation**

I. Introduction

Concurrent or distributed systems require efficient mechanisms to handle conflict between concurrent accesses to resources shared among several processes. Mutual exclusion locks are known as one of the most common techniques to solve such problems. Mutual exclusion guarantees that a process only can access the shared resources inside the critical section, and at most one process is allowed to enter the critical section at any time. In 1991, J. M. Mellor-Crummey and M. L. Scott proposed MCS mutual exclusion protocol [1]. Since then, MCS itself together with several variants of it were implemented and used in various environments. For example, variants of MCS have been used in Java Virtual Machines. The numerous implementations and extensive uses imply that the protocol is one of the most efficient and influential mutual exclusion algorithms.

However, a drawback of the MCS algorithm is that its exit segment pseudo-code is not wait-free. An algorithm is wait-free if every action of it by a process completes within a bound number of steps regardless of the behavior of other processes. In MCS, when a process leaving the critical section and releasing the lock, it needs to wait for the next process in the virtual queue to perform some steps. MCS also only works under the assumption that processes do not crash while acquiring/releasing the lock or while in the critical section. Failures or crashes, however, are often possible to happen in real systems. In [2], R. Dvir and G. Taubenfeld have proposed an extension to the original MCS to make the exit segment code wait-free. Based on the augmented MCS in [2], S. Dhoked and N. Mittal [3] have continously proposed another new version such that it is recoverable and wait-free. The recoverable property says that a process may crash at any point during its execution, but the protocol is able to recover and work as normal. Hereinafter, let us call the recoverable & wait-free version of MCS as RWfMCS.

In this paper, we formal specify RWfMCS and conduct model checking to confirm that when the protocol enjoys the mutual exclusion property and when it does not. Our experiments say that: (1) the protocol does not satisfy the mutual exclusion property if there is not any restriction to the occurrence of crashes; (2) the protocol enjoys the mutual exclusion property if crashes never happen; and (3) the protocol satisfies the mutual exclusion property if crashes have not occurred recently.

Formal specification of RWfMCS has a challenge in specifying dynamic memory allocation. When a process wants to enter the critical section, it first requests for allocating memory to initialize an empty node. Roughly speaking, nodes associated with processes are dynamically created. Unfortunately, modeling dynamic allocation in particular or dynamic systems in general is a non-trivial problem in formal method. P. C. Attie and N. A. Lynch [4] have addressed this problem and presented dynamic I/O automata, which is an extension of I/O automata to model and analyze dynamic systems. In this paper, we overcome that problem by providing a fixed list of "empty nodes" from the beginning. Every time a process requests for allocating memory to construct a new empty node, a top node of the list is extracted and used.

This research was partially supported by JSPS KAKENHI Grant Number JP19H04082.
DOI reference number: 10.18293/SEKE2021-065

II. Preliminaries

A Kripke structure K is $\langle S, I, T, P, L\rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a total binary relation over S, P is a set of atomic propositions and L is a labeling function whose type is $S \to 2^P$. Each element $(s, s') \in T$ is called a state transition from s to s' and T may be called the state transitions (with respect to K). For a state $s \in S$, $L(s)$ is the set of atomic propositions that hold in s. A path π is an infinite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots$ of states such that $s_i \in S$ and $(s_i, s_{i+1}) \in T$ for each i. Let π^i be $s_i, s_{i+1}, \ldots$ and $\pi(i)$ be s_i. Let P be the set of all paths. π is called a computation if $\pi(0) \in I$. Let C be the set of all computations.

The syntax of a formula φ in LTL for K is $\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi$, where $p \in P$. Let $\mathcal{F}$ be the set of all formulas in LTL for K. An arbitrary path $\pi \in P$ of K and an arbitrary LTL formula $\varphi \in \mathcal{F}$ of K, $K, \pi \models \varphi$ is inductively defined as $K, \pi \models \top$, $K, \pi \models p$ iff $p \in L(\pi(0))$, $K, \pi \models \neg\varphi_1$ iff $K, \pi \not\models \varphi_1$, $K, \pi \models \varphi_1 \wedge \varphi_2$ iff $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$, $K, \pi \models \bigcirc \varphi_1$ iff $K, \pi^1 \models \varphi 1$, and $K, \pi \models \varphi_1 \,\mathcal{U}\, \varphi_2$ iff 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, $K \models \varphi$ iff $K, \pi \models \varphi$ for each computation $\pi \in C$ of K. The temporal connectives $\bigcirc$ and $\mathcal{U}$ are called the next connective and the until connective, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond\varphi \triangleq \top \,\mathcal{U}\, \varphi$, and $\Box\varphi \triangleq \neg(\Diamond\neg\varphi)$. The temporal connectives $\Diamond$ and $\Box$ are called the eventually connective and the always connective, respectively.

In this paper, to express a state of S, we use an associative-commutative collection of name-value pairs. 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 $oc1, oc2, oc3$ be observable components, and then $oc1\ oc2\ oc3$ is the soup of those three observable components. A state is expressed as $\{oc1\ oc2\ oc3\}$. There are multiple possible ways to specify state transitions. In this paper, we use Maude [5], a programming/specification language based on rewriting logic, to specify them as rewrite rules. Maude makes it possible to specify complex systems flexibly and is also equipped with model checking facilities (a reachability analyzer and an LTL model checker). A rewrite rule starts with the keyword `rl`, followed by a label enclosed with square brackets and a colon, two patterns (terms that may contain variables) connected with =>, and ends with a full stop. A conditional one starts with the keyword `crl` and has a condition following the keyword `if` before a full stop. The following is a form of a conditional rewrite rule:

`crl` [lb] : l `=>` r `if` ... `/\` c_i `/\` ...

where lb is a label and c_i is part of the condition, which may be an equation $lc_i = rc_i$. The negation of $lc_i = rc_i$ could be written as $(lc_i =/= rc_i) =$ `true`, where `= true` could be omitted. If the condition ... `/\` c_i `/\` ... holds under some substitution σ, $\sigma(l)$ can be replaced with $\sigma(r)$.

Let `init` be the only initial state of K and φ be an LTL formula. Then, the Maude LTL model checker checks that K satisfies φ by the following command:

```
red modelCheck(init,φ) .
```

where `red` is an abbreviation of `reduce`. Executing this command, Maude will return either true if φ is satisfied, or a counterexample when φ is not satisfied.

III. The Recoverable Wait-free MCS protocol

The pseudo-code of RWfMCS protocol for each process i can be written as follows:

```
re   : if state[i] = LEAVE then goto ex1;
       else if state[i] = TRY and pred[i] = mine[i]
         then goto ex1;
       else if state[i] = FREE then {
         state[i] := INIT; mine[i] := null; }
en1  : if state[i] = INIT {
en2  :   if mine[i] = null then mine[i] := newNode();
en3  :   next_mine[i] := null; lock_mine[i] := true;
         pred[i] := mine[i]; state[i] := TRY; }
en4  : if state[i] = TRY then {
en5  :   if pred[i] = mine[i] {
en6  :     temp := FAS(tail, mine[i]);
en7  :     pred[i] := temp; | crash and goto re; }
en8  :   if pred[i] ≠ null
en9  :     if CAS(next_pred[i], null, mine_i)
en10 :       repeat while lock_mine[i];
cs   :   state[i] := InCS; }
ex1  : state[i] := LEAVE; CAS(tail, mine[i], null);
ex2  : if not CAS(next_mine[i], null, mine[i])
ex3  :   lock_next_mine[i] := false;
ex4  : state[i] = FREE; goto re;
```

RWfMCS maintains a queue of processes based on a linked list. That is why we call the queue is the virtual queue. Each element of the linked list is a node that contains the following two fields:

- *next*: stores the address of its successor (or next) node in the virtual queue if any and null otherwise.
- *lock*: stores a Boolean value. A process needs to spin to wait for its turn while trying to enter the critical section if its *lock* value is true.

$next_i$ and $lock_i$ can be regarded as the local variables of process i. RWfMCS uses the following global variables (i.e., shared among all processes):

- *pred*: an array where each element $pred[i]$ contains an address referring to the predecessor node of process i in the virtual queue if any and null otherwise.

- $mine$: an array where each element $mine[i]$ stores an address referring to the node associated with process i.
- $state$: an array where each element $state[i]$ receives one of the following values: FREE, INIT, TRY, InCS, or LEAVE.
- $tail$: contains the address of the last node in the virtual queue if the queue is not empty and null otherwise.

The algorithm consists of four segments code: recover section (label re), entering section (labels en1 to en10), critical section (label cs), and exiting section (labels ex1 to ex4). The function newNode() at the label en2 dynamically allocates memory, then initializes and returns an empty node. The body of the loop at the label en10 (between **repeat** and **while**) is empty. The algorithm uses the following two non-trivial atomic instructions:

- FAS (fetch-and-store): FAS(x, y) atomically does the following: x is set to y and the old value of x is returned.
- CAS (compare-and-swap): CAS(x, y, z) atomically does the following: if x equals y then x is set to z and true is returned, otherwise false is just returned.

We suppose that each process is located at one of the sixteenth labels, such as re, ex1, cs. Initially, each process is located at re; $next_i$ is null; $lock_i$ is false; $mine[i]$, $pred[i]$, and $tail$ are null; $state[i]$ is FREE. When a process wants to enter the critical section, it first moves to en1 from re.

To reduce the number of transitions so that to make it possible to conduct model checking later, at some labels (e.g., en3, ex1), we combine multiple assignments or instructions into only one transition. Furthermore, we suppose that crashes can only occur at en7. When a process is located at en7, it either performs assigning $temp$'s value to its $pred$ or crashes and goes back to the recover section non-deterministically. As mentioned in [3], the algorithm has only one "sensitive instruction" (i.e., the mutual exclusion property may not be satisfied if a process crashes immediately after executing this instruction) that is the one involving the FAS instruction at the label en6. That is the reason why we suppose that crashes can only be occurred at en7, after successfully performing the FAS instruction, but not yet storing the result to $pred[i]$. Whenever a process crashes, it loses the information about its local variables (i.e., $next$ and $lock$), but the shared variables do not be affected.

In the original MCS, when a process leaving the critical section and releasing the lock, it first tries to set $tail$ to null if $tail$ is still the process itself by using the CAS instruction. If $tail$ now refers to a different node (CAS return false), the virtual queue must contain at least another process requesting for the lock. In that case, the process releasing the lock needs to wait until its $next$ field contains a non-null reference (i.e., point to its successor in the virtual queue). However, the link between the process releasing the lock and its successor is created by the successor process, then the exit segment code of the original MCS algorithm is not wait-free. To overcome that problem, RWfMCS is augmented with some modifications at labels ex2 and en9. At label ex2, a process leaving the critical section sets the $next$ field of it to itself if the link from it to its successor process in the virtual queue has not been created yet in order to inform the next process in the queue that the lock is now free. While at label en9, by performing the CAS instruction, the next process in the queue checks the value of the $next$ field of its predecessor is null or not. If the value is null, indicating that the lock is now free, then the process can enter the critical section. On the other hand, it creates the link between its predecessor and itself, and spins to wait until its $lock$ becomes false.

IV. Formal specification the Protocol

In this paper, a state is expressed as a soup of observable components. To formalize RWfMCS as a Kripke structure K_{MCS}, we use the following observable components:

- (`tail` : p) - it says that $tail$ is p,
- (`pc`[p] : l) - it says that process `p` is located at label l,
- (`next`[p] : q) - it says that $next_p$ refers to q,
- (`lock`[p] : b) - it says that $lock_p$ is b,
- (`pred`[p] : q) - it says that $pred[p]$ refers to q,
- (`state`[p] : s) - it says that $state[p]$ is s,
- (`mine`[p] : q) - it says that $mine[p]$ refers to q,
- (`temp`[p] : q) - it says that $temp_p$ refers to q,

where p and q are two process IDs, l receives one of the sixteenth label values, b is a Boolean value, s receives one of five values of the $state$. Although in the pseudo-code, $temp$ is used as a temporary variable, in the specification, we explicitly use different $temp$ for each process to avoid undesirable behavior caused by jointly reading/writing $temp$.

Each state in S_{MCS} is expressed as $\{obs\}$, where obs is a soup of those observable components. If two processes `p1` and `p2` participate in RWfMCS, one initial state of I_{MCS} namely `init` is defined as follows:

```
{(tail: null) (pc[p1]: re) (pc[p2]: re)
(next[p1]: null) (next[p2]: null)
(lock[p1]: false) (lock[p2]: false)
(pred[p1]: null) (pred[p2]: null)
(mine[p1]: null) (mine[p2]: null)
(temp[p1]: null) (temp[p2]: null)
(state[p1]: FREE) (state[p2]: FREE)} .
```

There are seventeenth transitions for each process p:

- `rcv`: p performs the recover action (if crashed before) and moves to either en1 or ex1 from re,
- `chsta`: p checks the **if** condition at en1 and moves to either en2 or en4 from en1,
- `initmine`: p moves to en3 from en2,
- `init`: p moves to en4 from en3,
- `chsta2`: p checks the **if** condition at en4 and moves to either en5 or ex1 from en4,
- `chprd`: p checks the **if** condition at en5 and moves to either en6 or en8 from en5,

- `sttail`: p moves to en7 from en6,
- `stprd`: p moves to en8 from en7,
- `chprd2`: p checks the **if** condition at en8 and moves to either en9 or cs from en8,
- `stnxt`: p performs the CAS instruction at en9 and checks the returned value to move to either en10 or cs from en9,
- `chlck`: p tries to move to cs from en10,
- `exit`: p moves to ex1 from cs,
- `ststa`: p moves to ex2 from ex1,
- `stnxt2`: p performs the CAS instruction at ex2 and checks the returned value to move to either ex3 or ex4 from ex2,
- `stnxt2`: p moves to ex4 from ex3,
- `go2rcv`: p goes back to re from ex4,
- `crash`: when p is located at en7, p crashes and goes back to re.

Let `OCs` be a Maude variable of observable component soups, `P`, `Q`, `Q1` be Maude variables of process IDs, and `S` be Maude variable receives one of five values of the *state*. The rewrite rule `exit` is simply defined as follows:

```
rl [exit] : {(pc[P]: cs) (state[P]: S) OCs}
=> {(pc[P]: ex1) (state[P]: InCS) OCs} .
```

The rewrite rule says that when a process `P` is located at cs, `P` moves to ex1; and *state*[P] changes to InCS; other observable components do not change.

The rewrite rule `rcv` is defined as follows:

```
rl [rcv] : {(pc[P]: re) (state[P]: S)
(pred[P]: Q) (mine[P]: Q1) OCs}
=> {(pc[P]: (if S == LEAVE then ex1 else (
if S == TRY and Q == Q1 then ex1 else en1 fi)
fi)) (state[P]: (if S == FREE then INIT else S
fi)) (mine[P]: (if S == FREE then null else Q1
fi)) (pred[P]: Q) OCs} .
```

The rewrite rule says that when a process `P` located at re, if its *state* is LEAVE or its *state* is TRY and its *pred* equals to its *mine*, `P` then moves to ex1, otherwise, `P` moves to en1; if its *state* is FREE, its *state* changes to INIT and `mine[P]` is reset to null, otherwise, nothing changes; other observable components do not change.

One challenge we need to deal with during formally specifying RWfMCS is how to specify dynamic memory allocation. In the algorithm, when the function newNode() at label en2 is invoked, a new memory location is allocated from which an empty node is constructed and assigns to $mine[i]$. After the process i successfully enters the critical section, releases the lock, and goes back to the recover section, $mine[i]$ is reset to null (at label re). This assignment simply points $mine[i]$ to a null pointer, but the memory that contains the old node $mine[i]$ is still alive without any effect. Roughly speaking, the values returned by the function $newNode$ are different from time to time every time process i requests for allocating memory to construct a new empty node. Furthermore, resetting the value of $mine[i]$ to null does not affect the old value of $mine[i]$. It is, however, not simple to make the formal specification satisfying those behaviors.

To solve the problem of formally specifying dynamic memory allocation mentioned above, our solution is to provide a fixed list of "empty nodes" from the beginning. Every time a process requests for allocating memory to construct a new empty node (i.e., calls to the function newNode() at label en2), a top node of the list is extracted and used. When a process makes a request for a new node but the list of nodes now is empty, we let the process move to the terminal state in which the process spins there forever. We add one more observable component (`nodes`: lp), where lp is a list of process IDs, to represents the list of "empty nodes" used for dynamic allocation. Consequently, we add the following observable component to `init`:

```
(nodes: (q1 q2 q3 q4 q5 q6))
```

where `q`k is a process ID for each $k = 1, \ldots, 6$. Here we provide six "empty nodes" for dynamic allocation. The rewrite rule `initmine` now is defined as follows:

```
rl [initMine] : {(mine[P]: Q) (nodes: (Q1 LP))
(pc[P]: en2) OCs} => {(pc[P]: en3) (mine[P]:
(if Q == null then Q1 else Q fi)) (nodes:
(if Q == null then LP else (Q1 LP) fi)) OCs} .
```

where `LP` is a Maude variable whose value is a list of process IDs (possibly empty). The rewrite rule says that when a process `P` is located at en2 and `nodes` is not empty (i.e., consists of `Q1` and `LP`), `P` moves to en3; if `mine[P]` is null then two assignments are performed: assigning the top element of `nodes` (i.e., `Q1`) to `mine[P]`, and updating `nodes` by removing its top element.

We need to add a new rewrite rule to represent the transition when a process requests for allocating a new node but `nodes` now is empty. The rewrite rule is defined as follows:

```
rl [terminate] : {(pc[P]: en2) (mine[P]: null)
(nodes: empty) OCs} => {(pc[P]: terminal)
(mine[P]: null) (nodes: empty) OCs} .
```

where `terminal` is a new process location in addition to the sixteenth existing locations. When a process moves to `terminal`, it will stay there forever by the `stutter` rewrite rule that is defined as follows:

```
rl [stutter] : {(pc[P]: terminal) OCs}
=> {(pc[P]: terminal) OCs} .
```

The remaining transitions can be defined likewise.

V. Model checking

A. Model checking without any restriction to crashes

To model check that K_{MCS} satisfies some desired properties, we define P_{MCS} and L_{MCS}. P_{MCS} contains an atomic proposition namely `inCs` which takes a process IDs as its argument. L_{MCS} is initially specified as follows:

```
eq {(pc[P] : cs) OCs} |= inCs(P) = true .
eq {OCs} |= PROP = false [owise] .
```

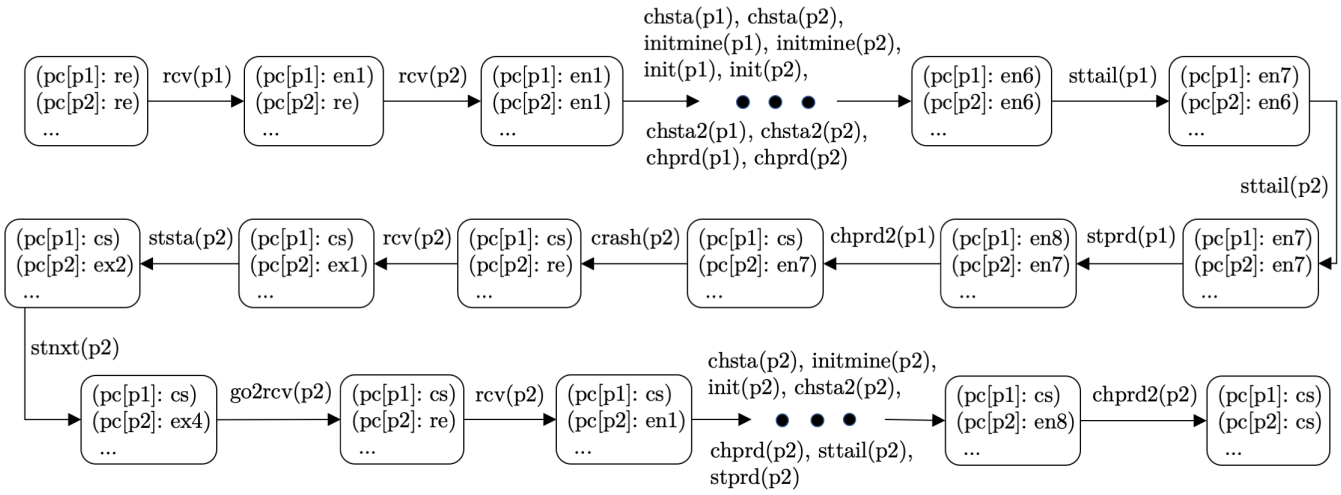

Figure 1. A counterexample shows that RWfMCS does not enjoy the mutual exclusion property if there is not any restriction to the occurrence of crashes

where `owise` is the abbreviation of otherwise, indicating that this equation will only be applied if all of the previous equations above it can not be applied. The equations say that `inCs(P)` holds in a state s iff s contains `(pc[P] : cs)`. We then specify the mutual exclusion property as the following LTL formula:

```
eq mutex = ([] ~(inCs(p1) /\ inCs(p2))) .
```

where `[]` is $\Box$, $\sim$ is $\neg$, and `/\` is $\wedge$. The equation (or formula) says that it is always the case such that `p1` and `p2` are not located at `cs` at the same time. We use Maude model checker to check that RWfMCS satisfies the mutual exclusion property or not by using the following Maude command:

```
red modelCheck(init,mutex) .
```

Unfortunately, a counterexample was found, which is visualized as in Fig. 1. Note that, the Figure does not show all observable components, but only depicts `pc[p1]` and `pc[p2]`, and uninteresting transitions are omitted (e.g., `chsta`, `initmine`). When the virtual queue consists of two processes `p1` and `p2` such that `p1` is located at cs, `p2` is located at en7, and $tail$ is `p2`, process `p2` crashes and goes back to re. Because $state$[p2] now is TRY, it then jumps to ex1, completes the exit segment in which it sets $tail$ to null by the CAS instruction. `p2` then tries to enter the critical section one more time. Since $tail$ is null now, if only transitions of `p2` are executed until it reaches en8 (without crash again), $pred$[p2] will be null. That time, p2 gets permission to directly enter cs, leading to the mutual exclusion property is not satisfied since there are two different processes `p1` and `p2` located at the critical section.

The first experiment says that if there is not any restriction to the occurrence of crashes, RWfMCS does not enjoy the mutual exclusion property. In the upcoming subsection, we report model checking under some assumptions of the occurrence of crashes.

B. Model checking under crash assumptions

First assumption: crashes never happen at all

We add the following observable component to keep track of the occurrence of crashes: (`crash`: b), where b is a Boolean value. Then, the rewrite rule `crash` is modified to become as follows:

```
rl [crash] : {(pc[P]: en7) (next[P]: Q)
(lock[P]: B) (crash: B1) OCs}
=> {(pc[P]: re) (next[P]: null)
(lock[P]: false) (crash: true) OCs} .
```

where `B` and `B1` are Maude Boolean variables. The rewrite rule says that when a process `P` is located at en7, it may crash then go back to re, `crash` is set to true, and `P` loses all information about its $next$ and $lock$. `crash` does not change in other transitions, and initially, it is set to false.

One more atomic proposition namely `crashed` is added into P_{MCS}. L_{MCS} is modified by adding the following equation before the existing `owise` statement at the end.

```
eq {(crash: true) OCs} |= crashed = true .
```

The equation says that `crashed` holds in a state s iff s contains `(crash: true)`. Since `crash` never can be changed back to `false` from `true`, we can say that when there is not any crash so far, `crashed` will not hold, otherwise, it will. We model check the mutual exclusion property under the assumption that crashes never happen at all by using the following command:

```
red modelCheck(init,([]~ crashed) -> mutex) .
```

No counterexamples were found. It took about 13 seconds for Maude to complete the model checking. Consequently, we can conclude that the protocol enjoys the mutual exclusion property if crashes never happen.

Second assumption: "crashes have not occurred recently"

The assumption that crashes never happen seems so strong, we should check it under a weaker assumption. In this section, we model check that RWfMCS enjoys the mutual exclusion property under the assumption: "crashes have not occurred recently". The key idea of this assumption is that after a crash, `crash` can be set back to false from true if all processes have *state* FREE. It means that after a crash, all requests for entering the critical section before are satisfied, implying the behavior of the protocol backs to normal as without failures. To check the condition whether all processes have *state* FREE, we need to introduce one more observable component namely (`noPsFree`: n), where n is a natural number. `noPsFree` maintains the number of processes that have *state* FREE. Initially, `noPsFree` is set to the number of processes participating in the protocol. When a process `P` is located at re and *state*[P] is FREE, the transition `rcv` will change its *state* to INIT, then `noPsFree` needs to be decreased by one. Thus, the rewrite rule `rcv` is modified to become as follows:

```
rl [rcv] : {(pc[P]: re) (state[P]: S)
(pred[P]: Q) (mine[P]: Q1) (noPsFree: N) OCs}
=> {(pc[P]: (if S == LEAVE then ex1 else (
if S == TRY and Q == Q1 then ex1 else en1 fi)
fi)) (state[P]: (if S == FREE then INIT else S
fi)) (mine[P]: (if S == FREE then null else Q1
fi)) (noPsFree: (if S == FREE then dec(N)
else N fi)) (pred[P]: Q) OCs} .
```

where `N` is a Maude variable of natural numbers, and `dec(N)` is a function that decreases `N` by one.

`noPsFree` needs to be increment when a process updates its *state* to FREE. In the algorithm, transition `go2rcv` is the only one that can change the *state* of a process to FREE. Therefore, the rewrite rule `go2rcv` is revised as follows:

```
rl [go2rcv] : {(pc[P]: ex4) (state[P]: S)
(noPsFree: N) (crash: B) OCs} => {(pc[P]: re)
(state[P]: FREE) (noPsFree: s(N)) (crash:
(if s(N) == 2 then false else B fi)) OCs} .
```

In addition to updating `noPsFree`, the rewrite rule also sets the value of `crash` back to true if the value of `noPsFree` after increasing is equal to the total number of processes participating in the protocol (i.e., 2 in our case).

One more atomic proposition namely `ncr` (not recently crash) is added into P_{MCS} to express the assumption that crashes have not occurred recently. L_{MCS} is modified by adding the following equation before the existing `owise` statement at the end.

```
eq {(crash: false) OCs} |= nrc = true .
```

The equation says that `nrc` holds in a state s iff s contains `(crash: false)`. We model check the mutual exclusion property under the assumption that crashes have not occurred recently by using the following Maude command:

```
red modelCheck(init,
  [] (nrc -> (~ (inCs(p1) /\ inCs(p2))))) .
```

No counterexamples were found. It took about 5 minutes for Maude to complete the model checking. Consequently, we can conclude that the protocol enjoys the mutual exclusion property if crashes have not occurred recently.

VI. Conclusion

We have presented model checking the recoverable wait-free version of MCS protocol. The recoverable property indicating that the protocol still works under the assumption that crashes may occur during the execution of processes. The wait-free property says that exit segment code of the new version is wait-free (i.e., always completes in a finite step regardless of the behavior of other processes). The first experiment shows a counterexample in which the protocol does not satisfy the mutual exclusion property. Analyzing the counterexample, we could understand the scenario leading to two different processes located at the critical section at the same time if crashes are allowed to happen without any restriction. The second experiment says that the protocol enjoys the mutual exclusion property if crashes never happen. The last experiment says that the protocol enjoys the mutual exclusion property if crashes have not occurred recently. We have also described the challenge of how to formally specify dynamic allocation and presented our solution to solve that problem during formal specifying the protocol.

One piece of our future work is to model check the protocol satisfies the lock-out freedom property in particular or other liveness properties in general. Model checking such properties usually requires some fairness assumptions. However, the formula to model check often becomes very complicated if some fairness assumptions are included, leading to the model checker could not terminate after a reasonable amount of time. One possible way to make it feasible is by using the technique presented in [6].

References

[1] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[2] R. Dvir and G. Taubenfeld, "Mutual exclusion algorithms with constant RMR complexity and wait-free exit code," in *OPODIS 2017*, ser. LIPIcs, vol. 95, 2017, pp. 17:1–17:16.

[3] S. Dhoked and N. Mittal, "An adaptive approach to recoverable mutual exclusion," in *PODC 2020*. ACM, 2020, pp. 1–10.

[4] P. C. Attie and N. A. Lynch, "Dynamic input/output automata: A formal and compositional model for dynamic systems," *Inf. Comput.*, vol. 249, pp. 28–75, 2016.

[5] M. Clavel, et al., Ed., *All About Maude*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[6] K. Ogata, "A divide & conquer approach to liveness model checking under fairness & anti-fairness assumptions," *Frontiers Comput. Sci.*, vol. 13, no. 1, pp. 51–72, 2019.