Analysis of the Suzuki-Kasami Algorithm with the Maude Model Checker

Kazuhiro Ogata
NEC Software Hokuriku, Ltd.
ogatak@acm.org

Kokichi Futatsugi
School of Information Science, JAIST
kokichi@jaist.ac.jp

Abstract

We report on a case study in which the Maude model checker has been used to analyze the Suzuki-Kasami distributed mutual exclusion algorithm with respect to the mutual exclusion property and the lockout freedom property. Maude is a specification and programming language/system based on membership equational logic and rewriting logic, equipped with model checking facilities. Maude allows users to use abstract data types, including inductively defined ones, in specifications to be model checked, which is one of the advantages of the Maude model checker. Hence, queues, which are used in the case study, do not have to be encoded in more basic data types. In the case study, the Maude model checker has found a counterexample that the algorithm is lockout free, which has led to one possible modification that makes the algorithm lockout free.

Keywords: counterexample, lockout freedom property, model checking, mutual exclusion property, rewriting logic.

1. Introduction

Maude [1, 2] (see `http://maude.cs.uiuc.edu/`) is a specification and programming language/system based on membership equational logic and rewriting logic. Data types are written in membership equational logic, and state machines (or transition systems) are written in rewriting logic. Maude can be characterized by fast (AC-)rewriting and excellent reflective (meta-programming) facilities. It is also equipped with a model checker. The model checker [5, 6] is an on-the-fly explicit state model checker whose assertions are written in propositional LTL (Linear Temporal Logic). Unlike other existing model checkers such as SMV [8], SPIN [7] and SAL [3], the state space of a state machine to be model checked by the Maude model checker does not necessarily have to be finite; the reachable states should be finite. In addition, the Maude model checker allows us to use abstract data types, including inductively defined ones, in specifications to be model checked, while other existing model checkers do not; complex data types should be encoded in more basic data types, which can often result in a complicated and error-prone process. Hence, queues, which are used in the case study described in this paper, do not have to be encoded in more basic data types.

The Suzuki-Kasami algorithm [10] is a distributed mutual exclusion algorithm. We have analyzed the Suzuki-Kasami algorithm in a semi-formal way [9] with respect to the mutual exclusion property and the lockout freedom property. The lockout freedom property may be called the starvation freedom property. We have partly written proofs (or proof scores) in CafeOBJ [4], an algebraic specification language, to show that the algorithm has the two properties. In the analysis, we have found a hidden assumption that is needed to make the algorithm lockout free. The assumption is that each node must try to enter its critical section infinitely often. In a distributed setting where some nodes try to enter their critical sections only finitely often, however, it is not guaranteed that the algorithm is lockout free.

In this paper, we report on the case study in which the Maude model checker has been used to analyze the Suzuki-Kasami algorithm with respect to the two properties. The Maude model checker has concluded that a model of the algorithm, whose reachable state space is finite, has the mutual exclusion property, but has found a counterexample that the algorithm has the lockout freedom property. The counterexample has led to one possible modification that makes the algorithm lockout free. The Maude model checker has finally concluded that a model of the modified algorithm, whose reachable state space is finite, has both mutual exclusion and lockout freedom properties. The modification makes the algorithm lockout free even if some nodes try to enter their critical sections only finitely often.

The rest of the paper is organized as follows. Section 2 describes the Suzuki-Kasami algorithm. Section 3 makes a mathematical model of the algorithm as a transition system. Section 4 specifies the transition system in Maude. Section 5 model checks the transition system in which two nodes are involved and each node makes at most two requests with respect to both mutual exclusion and lockout freedom properties. In the section, a possible modification

Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)
0-7695-2465-6/05 $20.00 © 2005 IEEE

to make the algorithm lockout free is proposed, which has been guided by counterexamples generated by the Maude model checkers. Section 6 finally concludes the paper.

2. The Suzuki-Kasami Algorithm

Let us consider a computer network consisting of a fixed number, say $N\ (\geq 1)$, of nodes. The 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.

The distributed mutual exclusion problem is to solve a mutual exclusion requirement for such a computer network, namely to allow at most one node to stay in its critical section at any given moment. The Suzuki-Kasami algorithm [10] is a distributed algorithm solving the problem. The algorithm may be called SKDMXA in this paper. The basic idea in the algorithm is to transfer the privilege for entering the critical sections. The node that owns the privilege is only allowed to enter its critical section. Figure 1 shows the algorithm for node $i \in \{1,2,\ldots,N\}$ in a traditional style.

requesting and *have_privilege* are Boolean variables. *requesting* 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 integers denoting node identifiers. It contains identifiers of nodes that wait to enter their critical sections. *ln* and *rn* are integer arrays of size N. *ln*[j] for each node $j \in \{1,2,\ldots,N\}$ is the sequence number of the node j's request granted most recently. *rn*[j] for each node $j \in \{1,2,\ldots,N\} - \{i\}$ records the largest request number ever received from node j. Node i uses *rn*[i] to generate the sequence numbers of its own requests. For each node $i \in \{1,2,\ldots,N\}$, initially, *requesting* is false, *have_privilege* is true if $i = 1$ and false otherwise, *queue* is empty, and *ln*[j] and *rn*[j] for each $j \in \{1,2,\ldots,N\}$ are 0.

There are two kinds of messages exchanged by nodes: *request* and *privilege* messages. A *request* message consists of an integer (a node identifier) and a sequence number, and a *privilege* message consists of a queue of integers (node identifiers) and an integer array of size N. A *request* message is used to let other nodes know that the sender wants to enter its critical section, and a *privilege* messages is used to transfer the privilege to another node.

If node i wants to enter its critical section, it first calls its own procedure *P1*, which first sets *requesting* 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

```
procedure P1;
begin
 requesting := true;
 if not have_privilege then
 begin
  rn[i] := rn[i] + 1;
  for all j in {1,2,...,N} − {i} do
   send request(i,rn[i]) to node j;
  wait until privilege(queue,ln) is received;
  have_privilege := true
 end;
 Critical Section;
 ln[i] := rn[i];
 for all j in {1,2,...,N} − {i} do
  if not in(queue,j) and (rn[j] = ln[j] + 1) then
   queue := put(queue,j);
 if queue ≠ empty then
 begin
  have_privilege := false;
  send privilege(get(queue),ln) to node top(queue)
 end;
 requesting := false
end;

{ requesting(j,n) is received; P2 is indivisible }
procedure P2;
begin
 rn[j] := max(rn[j],n);
 if have_privilege and not requesting
                  and (rn[j] = ln[j] + 1) then
 begin
  have_privilege := false;
  send privilege(queue,ln) to node j
 end
end;
```

Figure 1. The Suzuki-Kasami algorithm

it receives a privilege message *privilege*(*queue*,*ln*), it enters the critical section. When it finishes executing the critical section, it sets *ln*[i] to its current sequence number *rn*[i], which means that the current request has been granted, and updates *queue*, namely that identifiers of nodes that want to enter their critical sections and are not in the queue yet 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*(*get*(*queue*),*ln*) to the node found in the front of the queue, where *get*(*queue*) is the queue obtained by deleting the top element from *queue*. Otherwise, node i keeps the privilege. Finally node i sets *requesting* to false and leaves procedure *P1*.

Whenever *request*(j,n) is delivered to node i, node i ex-

Table 1. 13 parameterized transitions for each node $i \in \{1, 2, \ldots, N\}$

transition	label	the corresponding part in the algorithm
1. try_i	rem	calling procedure *P1*
2. set_req_i	l1	*requesting* := true;
3. $check\_priv_i$	l2	conditionally branching based on **not** *have_privilege*
4. $inc\_req\_no_i$	l3	*rn*[i] := *rn*[i] + 1;
5. $send\_req_i$	l4	executing one iteration of the first loop
6. $wait\_priv_i$	l5	**wait until** *privilege*(*queue*,*ln*) **is received**; *have_privilege* := true
7. $exit_i$	cs	leaving the critical section
8. $complete\_req_i$	l6	*ln*[i] := *rn*[i];
9. $update\_queue_i$	l7	executing one iteration of the second loop
10. $check\_queue_i$	l8	conditionally branching based on *queue* $\neq$ empty
11. $transfer\_priv_i$	l9	*have_privilege* := false; **send** *privilege*(*get*(*queue*),*ln*) **to** node *top*(*queue*)
12. $reset\_req_i$	l10	*requesting* := false
13. $receive\_req_i$	–	atomically executing procedure *P2*

ecutes its own procedure *P2*. But, 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 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.

3. Modeling the Suzuki-Kasami Algorithm

The algorithm is modeled as a transition system $\mathcal{S}_{\mathrm{SK}}^{M,N}$ that consists of a set $\mathcal{V}_{\mathrm{SK}}^{M,N}$ of variables, the initial condition $\mathcal{I}_{\mathrm{SK}}^{M,N}$ and a set $\mathcal{T}_{\mathrm{SK}}^{M,N}$ of transitions. The way of modeling the algorithm basically follows the one used in [9]. A transition may involve a condition called the effective condition. A transition can be executed only if the effective condition holds.

The set $\mathcal{V}_{\mathrm{SK}}^{M,N}$ of variables consists of eight parameterized variables and one non-parameterized variable. The variables are as follows:

1. $requesting_i$ corresponds to *requesting* in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
2. $have\_privilege_i$ corresponds to *have_privilege* in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
3. $queue_i$ corresponds to *queue* in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
4. ln_i corresponds to *ln* in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
5. rn_i corresponds to *rn* in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
6. idx_i corresponds to j in the algorithm for each node $i \in \{1, 2, \ldots, N\}$.
7. pc_i indicates which part of the algorithm node i will execute next for each node $i \in \{1, 2, \ldots, N\}$.
8. $num\_of\_req_i$ holds the number of requests node i has made for each node $i \in \{1, 2, \ldots, N\}$.
9. nw denotes the network; nw is a multiset of messages.

Whenever node i executes procedure *P1*, the sequence number $rn[i]$ is incremented. Therefore, in order to make the reachable states of the transition system $\mathcal{S}_{\mathrm{SK}}^{M,N}$ finite, the number of requests made by each node i should be finite, say $M\,(\geq 1)$. $num\_of\_req_i$ is used to allow node i to make at most M requests.

The set $\mathcal{T}_{\mathrm{SK}}^{M,N}$ of transitions consists of 13 parametrized transitions. Table 1 shows the 13 parameterized transitions for each node $i \in \{1, 2, \ldots, N\}$. The transitions exhaustively and exclusively correspond to parts of the algorithm as described in Table 1. The first 12 transitions denote procedure *P1* and the final one denotes procedure *P2*. The first 12 transitions are given labels (rem, l1, l2, l3, l4, l5, cs, l6, l7, l8, l9 and l10, respectively) as described in Table 1. pc_i is set to one of the labels.

The initial condition $\mathcal{I}_{\mathrm{SK}}^{M,N}$ is that $requesting_i$ is false, $have\_privilege_i$ is true if $i = 1$ and false otherwise, $queue_i$ is empty, each element of ln_i is 0, each element of rn_i is 0, idx_i is 1, pc_i is rem, $num\_of\_req_i$ is 0 for each $i \in \{1, 2, \ldots, N\}$, and nw is the empty multiset.

We describe (1) the effective condition of each transition and (2) how the transition changes variables in $\mathcal{V}_{\mathrm{SK}}^{M,N}$ when the transition is executed; variables that the transition

does not change may not be described explicitly. For each variable v, v and v' denote the values before and after a transition, respectively.

- try_i: (1) pc_i = rem. (2) If $num\_of\_req_i < M$, then pc'_i = l1 and $num\_of\_req'_i = num\_of\_req_i + 1$, and otherwise pc'_i = rem and $num\_of\_req'_i = num\_of\_req_i$.
- set_req_i: (1) pc_i = l1. (2) pc'_i = l2 and $requesting'_i$ = true.
- $check\_priv_i$: (1) pc_i = l2. (2) If $have\_privilege_i$ = true, then pc'_i = cs, and otherwise pc'_i = l3.
- $inc\_req\_no_i$: (1) pc_i = l3. (2) pc'_i = l4, $rn'_i[i] = rn_i[i] + 1$ and $idx'_i = 1$.
- $send\_req_i$: (1) pc_i = l4. (2) If $idx_i = N$, then pc'_i = l5 and $idx'_i = idx_i$, and otherwise pc'_i = l4 and $idx'_i = idx_i + 1$. Besides, a request message with i and $rn_i[i]$ is sent to node idx_i (is put into nw) unless $idx_i = i$.
- $wait\_priv_i$: (1) pc_i = l5 and there exists a privilege message with a queue Q of integers and an integer array LN of size N, whose destination is node i, in nw. (2) pc'_i = cs, $have\_privilege'_i$ = true, $queue'_i = Q$ and $ln'_i = LN$. Besides, the privilege message is removed from nw.
- $exit_i$: (1) pc_i = cs. (2) pc'_i = l6.
- $complete\_req_i$: (1) pc_i = l6. (2) pc'_i = l7, $ln'_i[i] = rn_i[i]$ and $idx'_i = 1$.
- $update\_queue_i$: (1) pc_i = l7. (2) If $idx_i = N$ then pc'_i = l8 and $idx'_i = idx_i$, and otherwise pc'_i = l7 and $idx'_i = idx_i + 1$. If $rn_i[idx_i] = ln_i[idx_i] + 1$ and idx_i is not in $queue_i$, then $queue'_i = put(queue_i, idx_i)$, and otherwise $queue'_i = queue_i$.
- $check\_queue_i$: (1) pc_i = l8. (2) If $queue_i$ is empty, then pc'_i = l10, and otherwise pc'_i = l9.
- $transfer\_priv_i$: (1) pc_i = l9. (2) pc'_i = l10 and $have\_privilege'_i$ = false. Besides, the privilege message with $get(queue_i)$ and ln_i is sent to node $top(queue_i)$ (is put into nw).
- $reset\_req_i$: (1) pc_i = l10. (2) pc'_i = rem and $requesting'_i$ = false.
- $receive\_req_i$: (1) there exists a request message with j ($\neq i$) and n in nw, whose destination is node i. (2) $rn'_i[j] = max(rn_i[j], n)$. Let C be $have\_privilege_i \wedge \neg requesting_i \wedge rn'_i[j] = ln_i[j] + 1$. If C is true, then $have\_privilege'_i$ = false and the privilege message with $queue_i$ and ln_i is sent to node n (is put into nw), and otherwise $have\_privilege'_i = have\_privilege_i$. Besides, the request message is removed from nw.

4. Specification of $\mathcal{S}_{\mathrm{SK}}^{M,N}$

4.1. Specifying Queues and Arrays

Maude allows us to use inductive data types in specifications to be model checked. Therefore, it is not necessary to encode queues in more basic types. Queues of natural numbers are specified in a module called `QUEUE`. In the module, we declare constant `empty` denoting the empty queue and operation `_|_` denoting the data constructor of nonempty queues as follows:

```
op empty :            -> Queue .
op _|_   : Nat Queue -> Queue .
```

Queue (declared in the module) and Nat are sorts denoting queues and natural numbers. An underscore `_` is the place where an argument is put. Given natural numbers `x`, `y` and `z`, term `x | y | z | empty` denotes the queue containing the natural numbers, whose top and last elements are `x` and `z`, respectively.

In module `QUEUE`, we declare operations `top`, `put`, `get`, `_\in_` and `empty?` corresponding to the usual functions for queues, and define them with equations. Given a queue q and a natural number x, `top`(q) is the top element of q, `put`(q, x) is the queue obtained by adding x into q at the end, `get`(q) is the queue obtained by deleting the top element from q, x `\in` q checks if q includes x, and `empty?`(q) checks if q is empty. The five operations are defined as follows:

```
op top    : Queue     -> Nat .
op put    : Queue Nat -> Queue .
op get    : Queue     -> Queue .
op _\in_  : Nat Queue -> Bool .
op empty? : Queue     -> Bool .
eq top(I | Q)    = I .
eq put(empty,I)  = I | empty .
eq put(J | Q,I)  = J | put(Q,I) .
eq get(I | Q)    = Q .
eq I \in empty   = false .
eq I \in (I | Q) = true .
eq I \in (J | Q) = I \in Q [owise] .
eq empty?(empty) = true .
eq empty?(I | Q) = false .
```

`I` and `J` are Maude variables of sort `Nat`, and `Q` is a Maude variable of sort `Queue`. Keyword `owise` stands for otherwise, which means that as long as the first and second equations of operation `_\in_` cannot be applied, then the third equation is used.

Since Maude does not provide arrays as a built-in data type, arrays should be specified. Arrays are modeled as multisets whose elements denote array elements. Integer arrays are specified in a module called `ARRAY`. In the module, sorts `AElm` and `Array` are declared, denoting elements and arrays, and `AElm` is declared as a subsort of `Array`. We

declare constant ia, and operations _:_ and _,_. Constant ia denotes an array whose elements are all 0, operation _:_ denotes the data constructor of array elements (the first and second arguments are an index and a value stored), and operation _,_ denotes the data constructor of arrays (it is given associativity, commutativity and ia as its identity). The constant and operations are declared as follows:

```
op ia  :              -> Array .
op _:_ : Nat Nat      -> AElm .
op _,_ : Array Array -> Array
                   [assoc comm id: ia] .
```

We also declare operations _[_] and _[_] :=_. Given an array a, an index i and a value x, term a[i] denotes the value stored at index i of array a and term a[i] := v denotes the array obtained by storing value v at index i of array a. The operations are defined in equations. The operations are declared and defined as follows:

```
op _[_]      : Array Nat     -> Nat .
op _[_] :=_ : Array Nat Nat -> Array .
eq ((I : X) , A)[I]         = X .
eq A[I]                     = 0 [owise] .
eq (((I : X) , A)[I] := Y) = (I : Y) , A .
eq (A[I] := Y)              = (I : Y) , A .
```

A is a Maude variable of sort Array, and I, J, X and Y are Maude variables of sort Nat.

4.2. Specifying the Network

Networks are modeled as a multiset of messages, specified in a module called NETWORK, which imports another module called MESSAGE where sort Message denoting messages is declared. In module NETWORK, we declare sort Network as a supersort of Message, constant void denoting the empty multiset (the empty network), and juxtaposition operation __ denoting the data constructor of nonempty multisets (nonempty networks). The juxtaposition operation is given associativity, commutativity and void as its identity. The constant and juxtaposition operation are declared as follows:

```
op void :                  -> Network .
op __   : Network Network -> Network
                   [assoc comm id: void] .
```

In module MESSAGE, we declare operation msg that is the data constructor of messages as follows:

```
op msg : Nat Request   -> Message .
op msg : Nat Privilege -> Message .
```

The operation takes a natural number and a message body as its arguments. The natural number denotes the destination of a message. Module MESSAGE imports two modules REQUEST and PRIVILEGE where we declare sorts Request and Privilege denoting requests and privileges, respectively. We declare operations req and priv in REQUEST and PRIVILEGE, respectively, as follows:

```
op req  : Nat Nat     -> Request .
op priv : Queue Array -> Privilege .
```

Operations req and priv are the data constructors of requests and privileges, respectively.

4.3. Specifying the Behavior of each Node

The behavior of each node is written in a module called SKDMXA, which imports two modules NETWORK and LABEL. In module LABEL, we declare sort Label denoting labels and 12 constants corresponding to the 12 labels.

In module SKDMXA, we declare sorts Var and Sys denoting variables and states of transition systems, respectively. Var is also declared as a subsort of Sys. States of transition systems are modeled as multisets of variables. To this end, the following constant and operation are declared:

```
op none :          -> Sys .
op __   : Sys Sys -> Sys
                   [assoc comm id: none] .
```

Constant none denotes the empty multiset (the empty state), which is the identity of juxtaposition operation __ that is the constructor of nonempty multisets (nonempty states). The juxtaposition operation is given associativity, commutativity and none as its identity.

In module SKDMXA, the following operations are declared:

```
op requesting[_]:_ : Nat Bool  -> Var .
op havePriv[_]:_   : Nat Bool  -> Var .
op rn[_]:_         : Nat Array -> Var .
op ln[_]:_         : Nat Array -> Var .
op queue[_]:_      : Nat Queue -> Var .
op idx[_]:_        : Nat Nat   -> Var .
op pc[_]:_         : Nat Label -> Var .
op numOfReq[_]:_   : Nat Nat   -> Var .
op nw:_            : Network   -> Var .
```

The operations correspond to the variables of transition system $\mathcal{S}_{\mathrm{SK}}^{M,N}$ described in Sect. 3, whose names are almost the same. For each of the operations, the last argument is the place where the value of the corresponding variable is put. For example, term requesting[1]: true means that the value of $requesting_1$ is true. We also declare two constants n and m, which are the number of nodes involved in SKDMXA and the number of requests made by each node.

The 13 kinds of transitions described in Sect. 3 are written as rewriting rules. Figure 2 shows the rewriting rules corresponding to $send\_req_i$, $wait\_priv_i$, $transfer\_priv_i$ and $receive\_req_i$. $send\_req_i$ is written using two rewriting rules. The remaining transitions are written as rewriting rules likewise.

```
rl [sendReq1] :
  (pc[I]: l4) (idx[I]: I)
  => (pc[I]: if I == n then l5 else l4 fi)
     (idx[I]: if I == n then I else I + 1 fi) .
crl [sendReq2] :
  (pc[I]: l4) (idx[I]: X) (rn[I]: RN) (nw: NW)
  => (pc[I]: if X == n then l5 else l4 fi)
     (idx[I]: if X == n then 1 else X + 1 fi)
     (rn[I]: RN) (nw: (msg(X,req(I,RN[I])) NW)) if X =/= I .

rl [waitPriv] :
  (pc[I]: l5) (havePriv[I]: F) (ln[I]: LN') (queue[I]: Q')
  (nw: (msg(I,priv(Q,LN)) NW))
  => (pc[I]: cs) (havePriv[I]: true) (ln[I]: LN) (queue[I]: Q)
     (nw: NW) .

rl [transferPriv] :
  (pc[I]: l9) (havePriv[I]: F) (ln[I]: LN) (queue[I]: Q) (nw: NW)
  => (pc[I]: l10) (havePriv[I]: false) (ln[I]: LN) (queue[I]: Q)
     (nw: (msg(top(Q),priv(get(Q),LN)) NW)) .

crl [receiveReq] :
  (requesting[I]: F) (havePriv[I]: F') (rn[I]: RN) (ln[I]: LN)
  (queue[I]: Q) (nw: (msg(I,req(J,X)) NW))
  => (requesting[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)
  if I =/= J /\
     Max := if (RN[J]) < X then X else RN[J] fi /\
     C := F' and not(F) and Max == (LN[J]) + 1 .
```

Figure 2. Rewriting rules for $send\_req_i$, $wait\_priv_i$, $transfer\_priv_i$ and $receive\_req_i$

5. Model Checking $\mathcal{S}_{\mathrm{SK}}^{M,N}$

The Maude model checker searches for all reachable states from a given initial state. We then define an initial state of transition system $\mathcal{S}_{\mathrm{SK}}^{M,N}$, which is written in a module called `SKDMXA-ISTATE` importing module `SKDMXA`. Module `SKDMXA-ISTATE` has operation `node` declared and defined as follows:

```
op node : Nat -> Sys .
eq node(I) = (numOfReq[I]: 0) (pc[I]: rem)
             (requesting[I]: false)
             (havePriv[I]: (I == 1))
             (rn[I]: ia) (ln[I]: ia)
             (queue[I]: empty) (idx[I]: 1) .
```

Term `node(I)` denotes the initial state of node `I`. The module also has constant `init` whose sort is `Sys`. The constant is defined as follows:

```
eq init = node(1) node(2) (nw: void) .
```

Constant `init` denotes an initial state of $\mathcal{S}_{\mathrm{SK}}^{2,2}$ where two nodes are involved. In the module, both constants `n` and `m` are defined as 2.

The properties to be checked are the mutual exclusion property and the lockout freedom property. We first write state predicates to write the properties as propositional LTL formulas, The state predicates are written in a module called `SKDMXA-PREDS` importing two modules `SKDMXA` and `SATISFACTION` (which is in the file `model-checker.maude` included in the Maude distribution). Module `SATISFACTION` imports module `LTL` where propositional LTL is specified. Module `LTL` declares two sorts `Prop` and `Formula` that denote state predicates and LTL formulas. `Prop` is a subsort of `Formula`. Module `SATISFACTION` declares sort `State` denoting states and operation `_|=_` that takes a state pattern and a state predicate to be defined as its arguments. In module `SKDMXA-PREDS`, we declare four operations `wait`, `crit`, `existPriv` and `existReq` denoting state predicates and define them as follows:

```
eq (pc[I]: l5) S |= wait(I)       = true .
eq (pc[I]: cs) S |= crit(I)       = true .
eq (nw: (msg(I,P) NW)) S
                 |= existPriv(I)  = true .
eq (nw: (msg(I,req(J,X)) NW)) S
                 |= existReq(I,J) = true .
```

`wait(I)` and `crit(I)` hold if node `I` is at `l5` and `cs`, respectively. `existPriv(I)` holds if there exists a privilege message in the network that is addressed to node `I`.

```
counterexample({... (pc[1]: rem) (pc[2]: rem) ..., 'try}
               {... (pc[1]: l1)  (pc[2]: rem) ..., 'try}
               {... (pc[1]: l1)  (pc[2]: l1)  ..., 'setReq}
               {... (pc[1]: l2)  (pc[2]: l1)  ..., 'setReq}
               {... (pc[1]: l2)  (pc[2]: l2)  ..., 'checkPriv1}
               {... (pc[1]: cs)  (pc[2]: l2)  ..., 'checkPriv2}
               {... (pc[1]: cs)  (pc[2]: l3)  ..., 'incRecNo}
               {... (pc[1]: cs)  (pc[2]: l4)  ..., 'sendReq2}
               {... (pc[1]: cs)  (pc[2]: l4)  ..., 'sendReq1}
               {... (pc[1]: cs)  (pc[2]: l5)  ..., 'exit}
               {... (pc[1]: l6)  (pc[2]: l5)  ..., 'completeReq}
               {... (pc[1]: l7)  (pc[2]: l5)  ..., 'updateQueue1}
               {... (pc[1]: l7)  (pc[2]: l5)  ..., 'updateQueue2}
               {... (pc[1]: l8)  (pc[2]: l5)  ..., 'checkQueue}
               {... (pc[1]: l10) (pc[2]: l5)  ..., 'resetReq}
               {... (pc[1]: rem) (pc[2]: l5)  ..., 'try}
               {... (pc[1]: l1)  (pc[2]: l5)  ..., 'setReq}
               {... (pc[1]: l2)  (pc[2]: l5)  ..., 'checkPriv1}
               {... (pc[1]: cs)  (pc[2]: l5)  ..., 'exit}
               {... (pc[1]: l6)  (pc[2]: l5)  ..., 'completeReq}
               {... (pc[1]: l7)  (pc[2]: l5)  ..., 'updateQueue1}
               {... (pc[1]: l7)  (pc[2]: l5)  ..., 'updateQueue2}
               {... (pc[1]: l8)  (pc[2]: l5)  ..., 'checkQueue}
               {... (pc[1]: l10) (pc[2]: l5)  ..., 'receiveReq}
               {... (pc[1]: l10) (pc[2]: l5)  ..., 'resetReq},
               {... (pc[1]: rem) (pc[2]: l5)  ..., 'try} )
```

Figure 3. An excerpted counterexample to the lockout freedom

`existReq(I,J)` holds if there exists a request message in the network that is sent to node `I` by node `J`.

The propositional LTL formulas denoting the two properties are written in a module called `SKDMXA-CHECK`. The module imports four modules `SKDMXA-ISTATE`, `SKDMXA-PREDS`, `MODEL-CHECKER` (which is in `model-checker.maude`) and `LTL-SIMPLIFIER` (which is in `model-checker.maude`). Module `MODEL-CHECKER` has operation `modelCheck` that takes two arguments denoting an initial state and a propositional LTL formula and returns the result of model checking. In `LTL-SIMPLIFIER`, operations and equations to simplify propositional LTL formulas are declared. It is optional to import `LTL-SIMPLIFIER`. In module `SKDMXA-CHECK`, we declare three constants `mutex`, `lofree` and `fairness` that denote the mutual exclusion property, the lockout freedom property and the weak fairness assumption given to transitions corresponding to the `waitPriv` and `receiveReq` rewriting rules, respectively. The operations are defined as follows:

```
eq mutex  = [] ~(crit(1) /\ crit(2)) .
eq lofree = (wait(1) |-> crit(1)) /\
            (wait(2) |-> crit(2)) .
eq fairness
 = (wait(1) /\ existPriv(1) |-> crit(1)) /\
   (existReq(1,2) |-> ~(existReq(1,2)))  /\
   (wait(2) /\ existPriv(2) |-> crit(2)) /\
   (existReq(2,1) |-> ~(existReq(2,1))) .
```

Operations `[]_`, `<>_` and `_|->_` denote Henceforth (Always), Eventually and Leads-to, respectively.

We use the Maude model checker to check if $\mathcal{S}_{\mathrm{SK}}^{2,2}$ in which two nodes are involved and each node makes at most two requests has the two properties by reducing the following two terms respectively:

1. `modelCheck(init,mutex)`
2. `modelCheck(init,(fairness => lofree))`

The Maude model checker concludes that $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has the mutual exclusion property but presents a counterexample that $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has the lockout freedom property.

A counterexample generated by the Maude model checker is the form `counterexample`(P, L), where P and L are lists of pairs, each of which consists of a state and the label of the rewriting rule applied to reach the state of the next pair. The next of the last pair of P is the first of L and the next of the last pair of L is the first of L. P corresponds to a finite path beginning in an initial state and leading to a loop represented by L. The loop represents a state or a transition sequence that violates a given LTL formula.

Figure 3 shows an excerpted counterexample that $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has the lockout freedom property; the counterexample has been generated by the Maude model checker. The counterexample says that node 2 never enters its critical section

```
crl [receiveReq] :
  (requesting[I]: F) (havePriv[I]: F') (rn[I]: RN) (ln[I]: LN)
  (queue[I]: Q) (nw: (msg(I,req(J,X)) NW))
  => (requesting[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)
  if I =/= J /\  L =/= l7 /\  L =/= l8 /\ L =/= l10 /\
     Max := if (RN[J]) < X then X else RN[J] fi  /\
     C := F' and not(F) and Max == (LN[J]) + 1 .
```

Figure 4. Finally revised rewriting rule corresponding to $receive\_req_i$

even though it wants to do. Taking a closer look at the counterexample lets us know that if node 1 receives a request message from node 2 at label `l10`, then node 2 may have to wait for a privilege message forever at label `l5`. This is because

1. when node 1 receives a request message from node 2 at label `l10`, node 1 does not transfer the privilege to node 2 because its `requesting` is true, and
2. if node 1 will not execute procedure *P1* anymore, then node 1 will never transfer the privilege to node2.

The counterexample suggests that each node should not receive any request messages at label `l10`. Therefore, `L =/= l10` is added to the condition of the rewriting rule labeled `receiveReq`. However, this is not enough. Model checking the revised $\mathcal{S}_{\mathrm{SK}}^{2,2}$ also generates a counterexample that the revised $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has the lockout freedom; the counterexample suggests that each node should not receive any request message at label `l8` either. Hence, `L =/= l8` is also added to the condition of the rewriting rule labeled `receiveReq`. Moreover, model checking the further revised $\mathcal{S}_{\mathrm{SK}}^{2,2}$ generates a counterexample that the further revised $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has the lockout freedom; the counterexample suggests that each node should not receive any request message at label `l7` either. Consequently, `L =/= l7` is added to the condition of the rewriting rule labeled `receiveReq` as well. The finally revised rewriting rule labeled `receiveReq` is shown in Fig. 4. The Maude model checker concludes that the finally revised $\mathcal{S}_{\mathrm{SK}}^{2,2}$ has both mutual exclusion and lockout freedom properties by reducing terms `modelCheck(init,mutex)` and `modelCheck(init,(fairness => lofree))`.

6. Conclusion

We have reported on the case study in which the Maude model checker has been used to analyze the Suzuki-Kasami algorithm with respect to the mutual exclusion property and the lockout freedom property. In the case study, the Maude model checker has presented a counterexample that the algorithm has the lockout freedom. The counterexample has led to a possible modification that makes the algorithm lockout free even if some nodes try to enter their critical sections only finitely often.

The case study also demonstrates that the state space of a state machine to be model checked by the Maude model checker does not necessarily have to be finite and inductive data types can be used in a specification of a state machine to be model checked by the Maude model checker. The state space of $\mathcal{S}_{\mathrm{SK}}^{2,2}$ is infinite because a queue of integers is used, although the reachable state space is finite, and queues of integers are inductively defined in the specification of $\mathcal{S}_{\mathrm{SK}}^{M,N}$.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming language in rewriting logic. *TCS*, 285(2):187–243, 2002.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *14th RTA*, LNCS 2706, pages 76–87. Springer, 2003.

[3] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *16th CAV*, LNCS 3114, pages 496–500. Springer, 2004.

[4] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[5] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *4th WRLA*, ENTCS 71. Elsevier, 2002.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *10th SPIN*, LNCS 2648, pages 230–234. Springer, 2003.

[7] G. J. Holzmann. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, 2004.

[8] K. L. McMillan. *Symolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.

[9] K. Ogata and K. Futatsugi. Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm. In *5th FMOODS*, pages 181–195. Kluwer, 2002.

[10] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM TOCS*, 3(4):344–349, 1985.