Analysis of the Suzuki-Kasami Algorithm with SAL Model Checkers

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 SAL model checkers have been used to analyze the Suzuki-Kasami distributed mutual exclusion algorithm with respect to the mutual exclusion property and the lockout freedom property. SAL includes five different model checkers. In the case study, we have used two model checkers SMC (Symbolic Model Checker) and infBMC (infinite Bounded Model Checker). SMC has concluded that a finite-state model of the algorithm has the mutual exclusion property, but has found a counterexample to the lockout freedom property. The counterexample has led to one possible modification that makes the algorithm lockout free. We have also used infBMC to prove that an infinite-state model of the algorithm has the mutual exclusion property by k-induction.

Keywords: counterexamples, distributed algorithms, lockout freedom, model checkers, mutual exclusion.

1. Introduction

With the advance of computer hardware technology, efficient model checking algorithms[4] have been devised and then model checkers such as SMV[6] and SPIN[6] have been developed. Model checkers can be used to verify fully automatically that given systems have desired properties, provided that the systems should be modeled as finite-state machines. SAL[1, 2] is a toolkit for analyzing state machines, providing several tools. Unlike other existing model checkers, each of which usually implements one model checking algorithm, SAL includes five different model checkers. The five different model checkers are symbolic, bounded, infinite bounded, witness and explicit-state model checkers (SMC, BMC, infBMC, WMC and EMC, respectively). Users can select the most appropriate one among the five different model checkers for their problems.

The Suzuki-Kasami algorithm[9] is a distributed mutual exclusion algorithm. We have analyzed the algorithm in a semi-formal way[7] with respect to the mutual exclusion property and the lockout (starvation) freedom property. 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 a case study in which SAL model checkers have been used to analyze the Suzuki-Kasami algorithm with respect to the two properties. SMC has concluded that a finite-state model of the algorithm has the mutual exclusion property, but has found a counterexample to the lockout freedom property. The counterexample has led to one possible modification that makes the algorithm lockout free. The modification makes the algorithm lockout free even if some nodes try to enter their critical sections only finitely often. We have also used infBMC to prove that an infinite-state model of the algorithm has the mutual exclusion property by k-induction[3].

The rest of the paper is organized as follows. Section 2 outlines SAL. Section 3 describes the Suzuki-Kasami algorithm. Section 4 models the Suzuki-Kasami algorithm as a transition system. Section 5 specifies the transition system in SAL. Section 6 checks if a finite-state model of the Suzuki-Kasami algorithm has the mutual exlusion and lockout freedom properties with SMC. Section 7 verifies that an infinite-state model of the Suzuki-Kasami algorithm has the mutual exlusion property with infBMC by k-induction. Section 8 finally concludes the paper.

2. SAL: Symbolic Analysis Laboratory

SAL[1, 2] (see `sal.csl.sri.com`) provides the five different model checkers (SMC, BMC, infBMC, WMC and EMC). EMC is not provided by SAL 2.3, but will be included in a future release of SAL. SMC, BMC, WMC and EMC deal with state machines defined over finite data types, while infBMC can also handle infinite data types such as integers. SMC, BMC, infBMC and EMC checks if assertions written in LTL hold for state machines, while WMC can handle assertions in CTL. BMC uses a propositional SAT solver to find counterexamples no longer than

Proceedings of the 2005 The Fifth International Conference on Computer and Information Technology (CIT'05)
0-7695-2432-X/05 $20.00 © 2005 IEEE

some specified depth. It can also verify that finite state machines have LTL properties by k-induction[3]. By default, SAL uses ICS[5] (Integrated Canonizer and Solver; see `www.icansolve.com`) as its SAT solver. For infinite state machines, infBMC can do what BMC does.

The SAL language consists of a type system and four languages for expressions, transition relations, modules and contexts. The SAL's type system and expression language are similar to those of PVS[8] (see `pvs.csl.sri.com`). Transition relations can be specified using both guarded commands and SMV-style variable-wise invariants. A state machine is written as a module that can have parameters. A module basically consists of a section of variable declarations, an initialization section and a section of transition relations. Modules can be composed both synchronously and asynchronously to generate compound state machines. Contexts include type declarations, function definitions, modules and assertions. Contexts can have parameters that are type and variable declarations. The assertion language is not primitive in SAL but is defined in libraries associated with the model checker concerned.

3. 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 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[9] 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. 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 either wants to enter or stays in 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* records the largest request number ever received from each one of the other nodes. 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.

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 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], indicating 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

```
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

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 executes 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, neither wants to enter nor stays in 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.

4 Modeling SKDMXA

The algorithm is modeled as a transition system $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ that consists of a set $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$ of variables, the initial condition $\mathcal{I}_{\mathrm{SK}}^{L,M,N}$ and a set $\mathcal{T}_{\mathrm{SK}}^{L,M,N}$ of transitions. The way of modeling the algorithm basically follows the one used in [7]. A transition may involve a condition called the effective condition. A transition can be executed only if the effective condition holds.

$\mathcal{V}_{\mathrm{SK}}^{L,M,N}$ includes $requesting_i$ corresponding to *requesting*, $have\_privilege_i$ corresponding to *have_privilege*, $queue_i$ corresponding to *queue*, ln_i corresponding to *ln*, and rn_i corresponding to *rn*, for each node $i \in \{1,2,\ldots,N\}$. $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$ also includes idx_i corresponding to j, pc_i, which indicates which part of the algorithm node i is about to execute, and $num\_of\_req_i$, which holds the number of requests node i has made, for each node $i \in \{1,2,\ldots,N\}$. In addition, $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$ includes variables denoting the network. How to model the network in SAL will be described in the coming section.

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

$\mathcal{T}_{\mathrm{SK}}^{L,M,N}$ contains 13 transitions, which are try_i, set_req_i, $check\_priv_i$, $inc\_req\_no_i$, $send\_req_i$, $wait\_priv_i$, $exit_i$, $complete\_req_i$, $up\,date\_queue_i$, $check\_queue_i$, $transfer\_priv_i$, $reset\_req_i$ and $receive\_req_i$ for each node $i \in \{1,2,\ldots,N\}$. The transitions exhaustively and exclusively correspond to parts of the algorithm. The first 12 kinds of transitions are given labels (rem, l1, l2, l3, l4, l15, cs, l6, l7, l8, l9 and l10, respectively). pc_i is set to one of the labels.

We describe (1) the effective condition of each transition and (2) how the transition changes variables in $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$ when the transition is applied; 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 the network) 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 the network. (2) pc_i' = cs, $have\_privilege_i'$ = true, $queue_i' = Q$ and $ln_i' = LN$. Besides, the privilege message is removed from the network.
- $exit_i$: (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 $top(queue_i)$.
- $reset\_req_i$: (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 the network, 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, and otherwise $have\_privilege_i' = have\_privilege_i$. Besides, the request message is removed from the network.

$\mathcal{I}_{\mathrm{SK}}^{L,M,N}$ 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 and $num\_of\_req_i$ is 0 for each $i \in \{1,2,\ldots,N\}$. In addition, variables denoting the network are set to values denoting the empty network.

5. Specification of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ in SAL

We describe how to specify $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ in SAL, specifically how to encode queues in more basic types, how to model the network and how to specify the behavior of each node.

5.1. Encoding Queues

Since recursive data types cannot be used in SAL specifications to be model checked, queues should be encoded in more basic types. Arrays and integers are used to encode queues. Type `Queue` is declared as a record type whose fields are `data` and `tl` as follows:

```
Queue: TYPE =
 [# data: ARRAY Queue_Idx OF Node_Id,
    tl: Queue_Idx #];
```

Types `Node_Id` and `Queue_Idx` are declared as bounded integers `[1..N]` and `[0..(L+1)]`, respectively. `N` is the number of nodes involved in the algorithm and `L` is the capacity of each queue.

Given an instance `q` of `Queue`, the `L` spaces from `q.data[1]` to `q.data[L]` are used to store queue elements; `q.data[1]` is always the top element of the queue if the queue is not empty; `q.tl` points the space where an element will be put next if the queue is not full and therefore `q.data[q.tl - 1]` is always the last element of the queue if the queue is not empty. We declare 9 functions related to `Queue`, which are `new_empty_queue` returning the empty queue, `full?` checking if a given queue is full, `empty?` checking if a given queue is empty, `in?` checking if a given queue includes a given element, `in_aux?` that is an auxiliary function for `in?`, `top` returning the top element of a given queue, `put` returning the queue obtained by putting a given element into a given queue at the end, `get` returning the queue obtained by deleting the top element of a given queue, and `get_aux` that is an auxiliary function for `get`.

In this paper, we only show the definitions of `in_aux?`, `in?`, `get_aux` and `get`:

```
in_aux?(queue : Queue, node : Node_Id,
        i : Queue_Idx): BOOLEAN =
  IF i = L+1 THEN false
  ELSIF queue.data[i] = node
     THEN i < queue.tl
  ELSE in_aux?(queue, node, i+1)
  ENDIF;

in?(queue : Queue,
    node : Node_Id): BOOLEAN =
  in_aux?(queue, node, 1);

get_aux(queue : Queue,
        i : Queue_Idx): Queue =
  IF i = L+1
  THEN queue WITH .tl := (queue.tl - 1)
  ELSE get_aux(queue WITH .data[i] :=
                       queue.data[i+1], i+1)
  ENDIF;

get(queue : Queue): Queue =
  IF empty?(queue)
  THEN queue ELSE get_aux(queue, 1) ENDIF;
```

Given an array a, an index i and a value v, the expression a `WITH` $.i := v$ denotes the array obtained by assigning v to the ith place of a.

In order to use a function in a SAL specification to be model checked, SAL must be able to prove the function terminating. If a function has a conditional choice (`IF THEN ELSE ENDIF`) statement whose condition is $L = R$ such that both L and R include variables whose values can be determined only at runtime, then SAL cannot be able to prove the function terminating. Therefore, SAL cannot prove the following definition of `in_aux?` (which seems more straightforward than that shown above) terminating:

```
in_aux?(queue : Queue, node : Node_Id,
        i : Queue_Idx): BOOLEAN =
  IF i = queue.tl THEN false
  ELSIF queue.data[i] = node THEN true
  ELSE in_aux?(queue, node, i+1)
  ENDIF;
```

In the first definition of `in_aux?`, since the value of `L` is given in advance and is not mutable, the value of `L+1` can be determined statically.

5.2. Modeling the Network

For each kind of message and each ordered (i, j) pair of nodes, we use a cell with which node i sends to node j one message of this kind. For each kind of message, we then use an array whose elements are arrays of messages of this kind. Since there are two kinds of messages, request and privilege messages, we use two such arrays to model the network.

In a module called `node` where the behavior of each node is written, we use two global variables `reqmedium` and `privmedium` to denote the network. The type of `reqmedium` is

```
ARRAY Node_Id OF ARRAY Node_Id OF ReqMsg
```

`reqmedium[i][j]` denotes the cell with which node `i` sends a request message to node `j`. The type of `privmedium` is

```
ARRAY Node_Id OF ARRAY Node_Id OF PrivMsg
```

`privmedium[i][j]` denotes the cell with which node `i` sends a privilege message to node `j`.

Types `ReqMsg` and `PrivMsg` denote cells with which messages are transferred. `ReqMsg` is declared as follows:

```
[# new: BOOLEAN, req: Request #]
```

`PrivMsg` is declared as follows:

```
send_req:
  pc = l4 --> pc' = IF idx = N THEN l5 ELSE l4 ENDIF;
              reqmedium'[i][idx]
              = (# new := true,
                   req := (# node := i, no := rn[i] #) #);
              idx' = IF idx = N THEN idx ELSE idx + 1 ENDIF

([] (j : Node_Id):
receive_req:
  reqmedium[j][i].new AND NOT(j = i)
  --> reqmedium'[j][i] = (# new := false,
                            req := (# node := 1, no := 0 #) #);
      rn'[j] = IF rn[j] < reqmedium[j][i].req.no
               THEN reqmedium[j][i].req.no
               ELSE rn[j] ENDIF;
      have_privilege'
      = IF have_privilege AND NOT(requesting) AND rn'[j] = ln[j] + 1
        THEN false ELSE have_privilege ENDIF;
      privmedium'[i][j]
      = IF have_privilege AND NOT(requesting) AND rn'[j] = ln[j] + 1
        THEN (# new := true,
                priv := (# queue := queue, done := ln #) #)
        ELSE privmedium[i][j] ENDIF)
```

Figure 2. Guarded commands corresponding to $send\_req_i$ and $receive\_req_i$

```
mutex: THEOREM
  system |- G(FORALL (i : Node_Id, j : Node_Id):
               (pc[i] = cs AND pc[j] = cs) => (i = j));

lofree: THEOREM
  system |- (FORALL (i : Node_Id, j : Node_Id):
              G((G(pc[i] = l5 AND
                   (EXISTS (k : Node_Id):
                     privmedium[k][i].new AND NOT(k = i)))
                 => F(pc[i] = cs)) AND
                (G(reqmedium[i][j].new) => F(NOT(reqmedium[i][j].new))) )
                => G(pc[i] = l5 => F(pc[i] = cs)) );
```

Figure 3. Assertions for the mutual exclusion and lockout freedom properties

```
[# new: BOOLEAN, priv: Privilege #]
```

For each record type, the second field holds a message to be transferred, and the first field indicates whether or not the cell contains a message.

Types `Request` and `Privilege` denote request and privilege messages, respectively. `Request` is declared as

```
[# node: Node_Id, no: Bnat #]
```

`Privilege` is declared as

```
[# queue: Queue,
   done: ARRAY Node_Id OF Bnat #]
```

Type `Bnat` is `[0..M]` and `M` is the number of requests made by each node.

5.3. Specifying the Behavior of each Node

The behavior of each node is written in module `node`, which has a parameter `i` whose type is `Node_Id`. In addition to the two global variables representing the network, namely `reqmedium` and `privmedium`, the following local variables are declared: `requesting` whose type is `BOOLEAN`, `have_privilege` whose type is `BOOLEAN`, `rn` whose type is `ARRAY Node_Id OF Bnat`, `ln` whose type is `ARRAY Node_Id OF Bnat`, `queue` whose type is `Queue`, `idx` whose type is `Node_Id`, `pc` whose type is `Label`, where Type `Label` is an enumeration type whose values correspond to the 12 labels such as rem and l1, and `num_of_req` whose type is `Bnat`. The variables correspond to $requesting_i$, $have\_privilege_i$, rn_i, ln_i, $queue_i$, idx_i, pc_i and num_of_req of transition system $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ described in Sect.4, respectively. The variables are initialized as described in Sect.4.

The 13 transitions described in Sect.4 are written using guarded commands. A guarded command is in the form *Label*: *Guard* `-->` *Effect*, where *Label* is the label given to the guarded command, which is an option, *Guard* is the condition, of the guard of the guarded command, and *Effect* is the effects (i.e. how to change the values of variables)

of the guarded command. The guarded commands corresponding to the 13 transitions are asynchronously composed to specify the behavior of each node as follows

$$\begin{array}{ll} & Label_1 : Guard_1 \texttt{ --> } Effect_1 \\ [] & Label_2 : Guard_2 \texttt{ --> } Effect_2 \\ & \cdots \\ [] & Label_{13} : Guard_{13} \texttt{ --> } Effect_{13} \end{array}$$

Some transition is written as multiple guarded commands that are asynchronously composed as follows:

$$([] \ (j : T) : L : G_j \texttt{ --> } E_j) ,$$

which is equivalent to

$$L : G_1 \texttt{ --> } E_1 \ [] \ldots [] L : G_N \texttt{ --> } E_N ,$$

where T is $[1 \,..\, N]$.

Figure 2 shows the guarded commands corresponding to $send\_req_i$ and $receive\_req_i$. $receive\_req_i$ is written as N guarded commands. The remaining transitions are written using guarded commands likewise.

Instances of module node are asynchronously composed to obtain the SAL specification of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$, which is written as follows:

```
system: MODULE
  = ([] (i : Node_Id): node[i])}.
```

6. Model Checking $\mathcal{S}_{\mathrm{SK}}^{2,2,2}$

By having each of N, L and M fixed, say 2, we use SMC to check if $\mathcal{S}_{\mathrm{SK}}^{2,2,2}$ has the mutual exclusion property and the lockout freedom property. The properties are written as the assertions labeled mutex and lofree, respectively, shown in Fig. 3. G and F are LTL temporal operators Henceforth (Always) and Eventually, respectively. We use the assumption that weak fairness is given to each of wait_priv and receive_req transitions to check the lockout freedom property. SMC concludes that mutex holds for $\mathcal{S}_{\mathrm{SK}}^{2,2,2}$, namely system such that $L = 2$, $M = 2$ and $N = 2$, but presents a counterexample to lofree.

A counterexample generated by a SAL model checker is composed of two lists P and L of pairs, each of which consists of a state (assignments to system variables) and the label of the guarded command 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 formula.

Figure 4 shows an excerpted counterexample to the lockout freedom generated by SMC. The counterexample says

```
Counterexample:
=======================
Path
=======================
Step 0:
--- System Variables (assignments) ---
... pc[1] = rem; pc[2] = rem; ...
-----------------------
Transition ... i = 1 ... try ...
-----------------------
Step 1:
--- System Variables (assignments) ---
... pc[1] = l1; pc[2] = rem; ...
-----------------------
Transition ... i = 1 ... set_req ...
-----------------------
...
Step 23:
--- System Variables (assignments) ---
... pc[1] = l10; pc[2] = l5; ...
-----------------------
Transition ... i = 1 ... receive_req ...
-----------------------
Step 24:
--- System Variables (assignments) ---
... pc[1] = l10; pc[2] = l5; ...
-----------------------
Transition ... i = 1 ... reset_req ...
-----------------------
Step 25:
--- System Variables (assignments) ---
... pc[1] = rem; pc[2] = l5; ...
=======================
Begin of Cycle
=======================
Step 25:
--- System Variables (assignments) ---
... pc[1] = rem; pc[2] = l5; ...
-----------------------
Transition ... i = 1 ... try ...
-----------------------
Step 26:
--- System Variables (assignments) ---
... pc[1] = rem; pc[2] = l5; ...
```

Figure 4. An excerpted counterexample to the lockout freedom

that if node 1 receives a request message from node 2 at l10, then node 2 may have to wait for a privilege message forever at l5. When node 1 receives a request message from node 2 at l10, node 1 does not transfer the privilege to node 2 because its requesting is true. If node 1 will not execute procedure *P1* anymore, node 1 will never transfer the privilege to node2 and node 2 will wait for a privilege message forever.

The counterexample suggests that each node should not receive any request messages at l10. Therefore, the condition NOT(pc = l10) is added to the guard of receive. But this is not sufficient. Two more performances of model checking have us notice that NOT(pc = l8) and NOT(pc = l7) should also be added to the guard. SMC

```
lemma1: LEMMA
 system |- G(FORALL (i : Node_Id):
             ((pc[i] = l2) => requesting[i]) AND
             ((pc[i] = l3) => requesting[i]) AND
             ((pc[i] = l4) => requesting[i]) AND
             ((pc[i] = l5) => requesting[i]) AND
             ((pc[i] = cs) => (requesting[i] AND have_privilege[i])) AND
             ((pc[i] = l6) => (requesting[i] AND have_privilege[i])) AND
             ((pc[i] = l7) => (requesting[i] AND have_privilege[i])) AND
             ((pc[i] = l8) => (requesting[i] AND have_privilege[i])) AND
             ((pc[i] = l9) => (requesting[i] AND have_privilege[i])) );

lemma2: LEMMA
 system |- G(FORALL (i : Node_Id, j : Node_Id, k : Node_Id, l : Node_Id):
             ((have_privilege[i] AND have_privilege[j]) => (i = j)) AND
             (have_privilege[i] => NOT(privmedium[j][k].new)) AND
             ((privmedium[i][j].new AND privmedium[k][l].new)
                                                 => (i = k AND j = l)) );
```

Figure 5. Lemmas

finally concludes that both `mutex` and `lofree` hold for the revised $\mathcal{S}_{\mathrm{SK}}^{2,2,2}$.

7. Verification of $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$ with k-induction

In $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ analyzed with SMC, we made the number M of requests made by each node finite to make the state space of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ finite. We relax the restriction, namely allowing each node to make an arbitrary number of requests. Let such $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ is represented by $\mathcal{S}_{\mathrm{SK}}^{L,\infty,N}$. infBMC can be used to analyze $\mathcal{S}_{\mathrm{SK}}^{L,\infty,N}$ whose (even reachable) state space is infinite. Let both of L and N be 2.

The property to be analyzed is the mutual exclusion property. We can use infBMC to search for a counterexample to `mutex` no longer than some specified depth, say 10. infBMC concludes that there are no counterexamples in the specified range.

We can also use infBMC to verify that $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$ has the mutual exclusion property with k-induction. To this end, first we should verify that two lemmas hold for $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$. The first lemma is written as the assertion labeled `lemma1` shown in Fig.5. infBMC can verify that `lemma1` holds for $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$ with 1-induction. The second lemma is written as the assertion labeled `lemma2` shown in Fig.5. infBMC can verify that `lemma2` holds for $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$ with 1-induction using `lemma1` as a lemma. Using the two lemmas, infBMC can verify that `mutex` holds for $\mathcal{S}_{\mathrm{SK}}^{2,\infty,2}$ with 1-induction.

The structure of the verification with k-induction would be useful when we verify that the infinite model involving an arbitrary number of nodes has the mutual exclusion property with a theorem prover such as PVS[8]. In other words, we would need the two lemmas to verify that $\mathcal{S}_{\mathrm{SK}}^{L,\infty,N}$ involving an arbitrary number N of nodes has the property. Actually, when we verified it in a semi-formal way[7], we used two lemmas that were almost the same as `lemma1` and `lemma2`.

8. Conclusion

We reported on a case study in which SAL model checkers, precisely SMC and infBMC, have been used to analyze the Suzuki-Kasami algorithm with respect to the mutual exclusion property and the lockout freedom property. In the case study, we have found out one possible modification that makes the algorithm lockout free even if some nodes try to enter their critical sections only finitely often.

References

[1] 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.

[2] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. CSL Technical Report SRI-CSL-01-02 (Rev. 2), August 2003.

[3] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *15th CAV*, LNCS 2392, pages 14–26. Springer, 2003.

[4] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.

[5] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *13th CAV*, LNCS 2102, pages 246–249. Springer, 2001.

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

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

[8] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th CADE*, LNAI 607, pages 748–752. Springer, 1992.

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

0-7695-2432-X/05 $20.00 © 2005 IEEE