2018 5th International Conference on Dependable Systems and Their Applications (DSA)

Model Checking of the Suzuki-Kasami Distributed Mutual Exclusion Algorithm with SPIN

Shouki Sakamoto, Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {s0710202, ogata}@jaist.ac.jp

***Abstract*—We report on a case study in which we have model checked some properties for the Suzuki-Kasami distributed mutual exclusion algorithm with SPIN. In the case study, two different system specifications of the algorithm were written in PROMELA and compared by model checking some properties for the two system specifications with SPIN. We also compare SPIN and Maude (and also SAL) to make the characteristics of SPIN clear.**

***Keywords*-SAL; Spin; Suzuki-Kasami Protocol; Maude; model checking**

I. Introduction

We have conducted an additional experiment of what is described in [1], which compares Maude [2] and SAL [3] by model checking some properties for the Suzuki-Kasami distributed mutual exclusion algorithm [4] (abbr. as the Suzuki-Kasami algorithm) with Maude and SAL. The additional experiment is to model check the same properties for the Suzuki-Kasami algorithm with SPIN [5]. A brief comparison of SPIN and Maude (and also SAL) is mentioned in [1]. In order to make the characteristics of those three model checkers (especially SPIN) clearer, however, we need to have a firsthand experience of SPIN. This is why we have conducted the additional experiment.

In the additional experiment, we first wrote the transition system modeling the Suzuki-Kasami algorithm in PROMELA that is the system specification language of SPIN. The PROMELA document is called SysSpec1. We next model checked the mutual exclusion property and the lockout freedom property for SysSpec1 with SPIN. SPIN concluded that the former property holds for SysSpec1, but found a counterexample of the latter property as Maude and SAL did. We modified the algorithm and SysSpec1 such that SPIN concludes that the two properties hold for the modified SysSpec1. Since it is possible to write the algorithm more directly in PROMELA, we then wrote the modified algorithm in PROMELA without explicitly considering the transition system. The PROMELA document is called SysSpec2. We then model checked the two properties for SysSpec2.

The study was mainly conducted by the first author as his minor-theme research supervised by the second author when the first author was a JAIST Master's student.

Among the lessons learned from the additional experiment are as follows:

1) Transition systems can be naturally written in PROMELA.
2) Channels can be used as the data structure queues.
3) It is straightforward for people who have some experience in programming to write algorithms directly in PROMELA.
4) An abstract mathematical model (a transition system) of an algorithm should be made so as to use SPIN efficiently.

Lesson 1 may seem strange because a PROMELA document corresponds to a transition system. The lesson says that it is possible to write a transition system in PROMELA such that the transition system coincides with the transition system to which the PROMELA document corresponds.

Since inductive data types (or structures) cannot be written in PROMELA, some compounded data structures such as lists should be encoded with more basic data types as we should do in SAL [1]. You do not need to encode queues, however, because channels, which are usually used to transfer messages, can be used as queues[1]. But, you need to specify the size of each channel in advance.

SysSpec2 was first written by the second author as his sub-theme project of JAIST in the first year of his master course. Before the sub-theme, he did not have any experiences in model checking, although he did in functional programming. SPIN can analyze SysSpec1 more efficiently than SysSpec2. This is because the latter has a larger number of states than the former. Lessons 3 & 4 say that non-experts can use SPIN for model checking, but they should learn some basic notions and techniques such as transition systems and how to model algorithms as transition systems so as to use SPIN efficiently.

The rest of the paper is organized as follows. Section II describes the Suzuki-Kasami algorithm. Section III describes the transition system modeling the algorithm. Section IV describes the system specification of the transition system in

[1]Some SPIN expert told us that this is a standard technique for SPIN users.

978-1-5386-9266-0/18/$31.00 ©2018 IEEE
DOI 10.1109/DSA.2018.00031

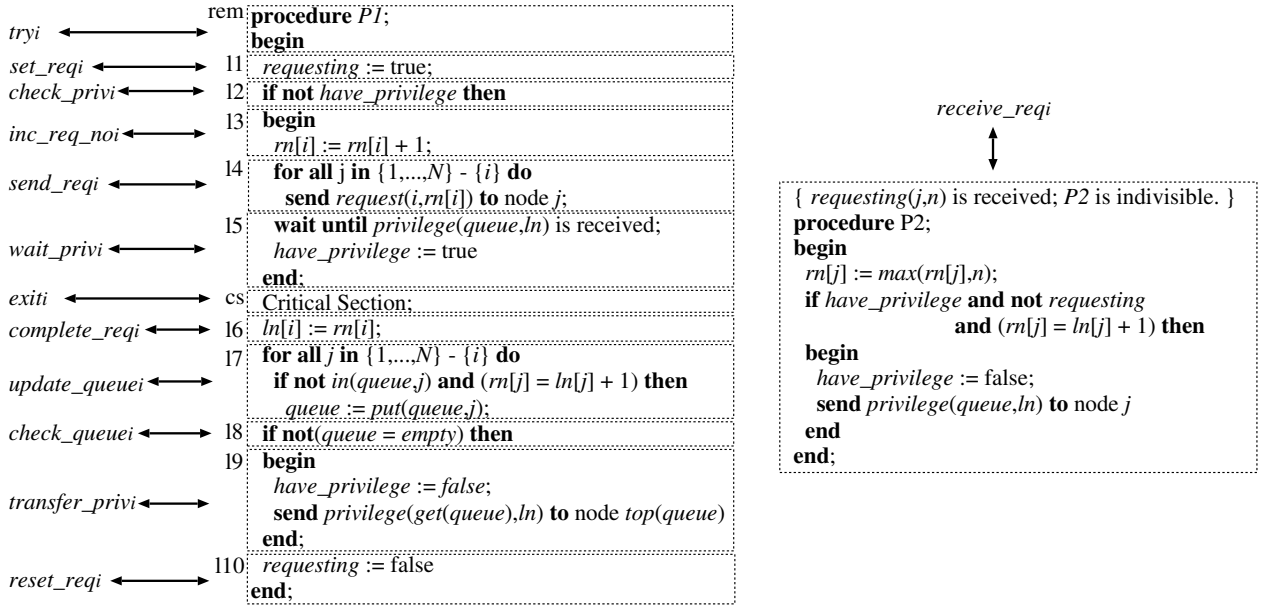

Figure 1. Correspondence between transitions and the Suzuki-Kasami algorithm

PROMELA. Section V describes model checking of the two properties for the system specification with SPIN. Section VI describes another system specification obtained by writing the modified algorithm directly in PROMELA. Section VII describes the characteristics of SPIN found through the case study. Section VIII concludes the paper.

We suppose that readers have some basic knowledge of PROMELA.

II. The Suzuki-Kasami Algorithm

The algorithm for node $i \in \{1, 2, \ldots, N\}$ in a traditional style is shown in Figure 1. The algorithm consists of the two procedures *P1* and *P2*. When node i wants to enter the critical section, it calls *P1*. When node i receives a request message, it calls *P2*, which should be executed atomically. The algorithm uses two Boolean variables *requesting* and *have_privilege*, one queue *queue* of node identifiers $(1, \ldots, N)$, and two integer arrays *ln* and *rn* of size N for each node. 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.

III. Modeling the Algorithm

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 [1]. L is the capacity of each queue, M is the number of requests made by each node, and N is the number of nodes. A transition has a condition such that a transition can be executed only if the condition holds.

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

$\mathcal{T}_{\mathrm{SK}}^{L,M,N}$ contains 13 transitions for each node i. The transitions exhaustively and exclusively correspond to parts of the algorithm. Figure 1 shows the correspondence between the 13 transitions and the Suzuki-Kasami algorithm. The 12 transitions that correspond to procedure *P1* are given labels (rem, l1, l2, l3, l4, l5, cs, l6, l7, l8, l9 and l10, respectively). pc_i is set to one of the labels.

$\mathcal{I}_{\mathrm{SK}}^{L,M,N}$ 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, 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.

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

Macros are used to define `L`, `M` and `N`, which are replaced with some small natural numbers and correspond to the parameters L, M and N of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$. Macros are also used

to define `rem`, `l1`, ..., `l10`, which are replaced with 0, 1, ..., 11, respectively, and correspond to the labels rem, l1, ..., l10.

The following structured data type is declared to represent some variables in $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$:

```
typedef Node {
  byte pc;
  bool have_privilege;
  bool requesting;
  byte rn[N];
  LN ln;
  chan queue = [L] of {byte};
}
```

`LN` is another structured data type, which is declared as follows:

```
typedef LN { byte ln[N]; }
```

Since arrays themselves cannot be transfered vis channels, the array `ln` is embedded in the structured data type. The following array variable `node` is declared:

```
Node node[N];
```

Each slot `node[`i`]` represents the node i's state. Since arrays are indexed from 0, we use $0, 1, \ldots, N-1$ as node identifiers instead of $1, 2, \ldots, N$.

The network is represented by channels. The following two arrays of channels are then declared:

```
chan req[N] = [L] of {byte,byte};
chan priv[N] = [L] of {chan,LN};
```

The channel `req[`i`]` is used to send a request message to node i, and the channel `priv[`i`]` is used to send a privilege message to node i.

The behavior (i.e. the transitions) of each node is written in PROMELA as follows:

```
proctype p(byte i) {
  do
  :: trans_1
       ⋮
  :: trans_m
  od
}
```

One transition whose condition holds is chosen nondeterministically and executed, which is repeated. Each transition should be atomically executed. Therefore, each $trans_k$ is written as follows:

```
atomic { cond_k -> effect_k }
```

$cond_k$ is the condition of $trans_k$ and $effect_k$ is the effect of $trans_k$. $effect_k$ consists of PROMELA statements.

Units of weak fairness in SPIN are processes. If a process remains executable from some time on under weak fairness, then the process will be eventually executed. When multiple transitions remain executable from some time on under weak fairness, however, some transitions may be never executed because one among such multiple transitions is chosen nondeterministically and executed.

In $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$, exactly one transition of the 12 transitions corresponding to *P1* for each node i is executable at any given moment. This is because their conditions do not overlap each other, depending on the different value of pc_i. But, the condition of $receive\_req_i$ overlap the conditions of the other transitions. Hence, two transitions, one of which is $receive\_req_i$, may become executable simultaneously. If all the 13 transitions for each node are written in one process, then $receive\_req_i$ may be never executed even under weak fairness. Therefore, we use two processes, which are instances of `proctype` declarations `p1` and `p2`, for each node i. The 12 transitions corresponding to *P1* are written in `p1`, and $receive\_req_i$ is written in `p2`. The `proctype` declarations `p1` and `p2` correspond to the procedures *P1* and *P2*, respectively.

The outline of `p1` is as follows:

```
proctype p1(byte i) {
  byte idx, req_num;
  do
  ...
  od
}
```

The local variables `idx` and `req_num` correspond to idx_i and $num\_of\_req_i$ in $\mathcal{V}_{\mathrm{SK}}^{L,M,N}$, respectively. The 12 transitions corresponding to procedure *P1* are written as option sequences of the `do` repetition structure. In this paper, the option sequences corresponding to $send\_req_i$ and $wait\_priv_i$ are only shown:

```
:: atomic { /* send_req_i */
     node[i].pc == l4
     ->
     if
     :: (idx < N && idx != i) ->
        req[idx] ! i, node[i].rn[i];
        idx++;
     :: (idx < N && idx == i) -> idx++;
     :: else ->
        node[i].pc = l5;
     fi;
   }

:: atomic { /* wait_priv_i */
     (node[i].pc == l5 &&
      priv[i]
        ? [node[i].queue, node[i].ln])
     ->
     node[i].pc = cs;
     priv[i]
```

```
       ? node[i].queue, node[i].ln;
     node[i].have_privilege = 1;
   }
```

p2 corresponding to *receive_req*$_i$ is as follows:

```
proctype p2(byte i) {
 byte j, n;
 do
 :: atomic { /* receive_req_i */
    req[i] ? [j, n]
    ->
    req[i] ? j, n;
    if
    :: node[i].rn[j] < n
       -> node[i].rn[j] = n;
    :: else -> skip;
    fi;
    if
    :: (node[i].have_privilege &&
        !(node[i].requesting) &&
        node[i].rn[j]
         == node[i].ln.ln[j] + 1)
      -> node[i].have_privilege = false;
         priv[j]
          ! node[i].queue, node[i].ln;
    :: else -> skip;
    fi
   }
 od
}
```

The PROMELA document of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ is called SysSpec1.

We have one more process that initializes some global variables and instantiates the two `proctype` declarations for each node. The process is as follows:

```
init {
 byte i;
 atomic{
  node[0].have_privilege = true;
  do
  :: i < N
    -> run p1(i);
       run p2(i);
       i++;
  :: else -> break;
  od
 }
}
```

Since the default initial values of (both scalar and array) variables are 0, which is equivalent to `false`, we only need to explicitly initialize

```
node[0].have_privilege
```

to `true`, which is equivalent to 1.

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

When the number N of nodes is fixed to 2, the mutual exclusion property is expressed as `[] !(q0 && q1)`, where qi is `(node[`i`].pc == cs)` for $i = 0, 1$. Both L and M are also fixed to 1. SPIN does not find any counterexamples of the property.

When the number N of nodes is fixed to 2, the lockout freedom property is expressed as `([] (p0 -> <> q0)) && ([] (p1 -> <> q1))`, where pi is `(node[`i`].pc == l1)` for $i = 0, 1$[2]. Both L and M are also fixed to 1. Weak fairness is used to model check the property because otherwise SPIN shows a non-fair computation path as a counterexample. SPIN finds a counterexample of the property, which is the same as what Maude and SAL found. According to the counterexample, $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ is modified. The modification is to strengthen the condition of *receive_req*$_i$ as follows:

```
(node[i].pc != l10 && node[i].pc != l8
 && node[i].pc != l7 && req[i] ? [j, n])
```

Then, SPIN does not find any counterexample of the property for the modified $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$.

VI. Another Specification of the Algorithm

It is possible to write the Suzuki-Kasami algorithm directly in PROMELA without considering $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$. We describe the system specification (called SysSpec2) obtained by writing the modified algorithm directly in PROMELA. The data structures and variables used in SysSpec2 are almost the same as those used in SysSpec1. Only the difference is as follows: the 12 labels `rem`, `l1`, ..., `l10` are used in SysSpec1, but the four labels `rs`, `ws`. `cs` and `ps` are used in SysSpec2. The behavior of each node is also specified as two `proctype` declarations `p1` and `p2`.

We show part of `p1`, which is as follows:

```
proctype p1(byte i) {
  byte idx, req_num;
loop:
  do
  :: req_num < M -> break;
  :: else -> skip;
  od;
  req_num++;
  node[i].requesting = true;
  node[i].pc = ws;
  if
  :: !node[i].have_privilege ->
     node[i].rn[i]++;
```

[2]In [1], the lockout freedom property is expressed as $\Box(pc_i = \mathrm{l5} \rightarrow \Diamond(pc_i = \mathrm{cs}))$ for each node i because l5 is the only place where nodes that want to enter their critical sections may be stuck. Precisely, however, the property is expressed as $\Box(pc_i = \mathrm{l1} \rightarrow \Diamond(pc_i = \mathrm{cs}))$ for each node i.

```
      idx = 0;
      do
      :: (idx < N && idx != i) ->
         req[idx] ! i, node[i].rn[i];
         idx++;
      :: (idx < N && idx == i) -> idx++;
      :: else -> break;
      od;
      priv[i] ? node[i].queue,node[i].ln;
      node[i].have_privilege = true;
   :: else -> skip;
   fi;
   node[i].pc = cs;
   /* Critical Section */
   node[i].pc = ps;
   node[i].ln.ln[i] = node[i].rn[i];
   ...
   node[i].requesting = false;
   node[i].pc = rs;
   goto loop;
}
```

`p2` used in SysSpec2 is almost the same as `p2` used in the modified SysSpec1. Only the difference is the condition of the option sequence used in `p2`. The condition used in SysSpec2 is as follows:

```
(node[i].pc != ps && req[i] ? [j, n])
```

SysSpec2 has the same `init` process as SysSpec1 has.

VII. Comparison

We describe the characteristics of SPIN found through the case study. We also report on experimental results on performance of model checking.

A. SPIN, Maude and SAL

Maude and SAL are compared in terms of six functionalities in [1]. We describe SPIN in terms of the six functionalities.

- Support of Inductive Data Types: Unlike Maude, SPIN does not support inductive data types. But, channels can be used as bounded queues. Therefore, we do not need to encode queues when the Suzuki-Kasami algorithm is written in PROMELA.
- Automatic Verification by k-induction: Unlike SAL, SPIN does not support it.
- Support of Fairness Assumptions: Unlike Maude and SAL, SPIN supports weak fairness whose units are processes.
- Support of Quantifiers: Unlike SAL, SPIN does not support quantifiers of property specifications.

One of the characteristics that SPIN has but neither Maude nor SAL has is as follows:

Table I
Experimental Results on Performance

(a) Model checking for the modified SysSpec1 with SPIN (exhaustive search)

	$\mathcal{S}_{\mathrm{SK}}^{1,1,2}$	$\mathcal{S}_{\mathrm{SK}}^{2,1,3}$	$\mathcal{S}_{\mathrm{SK}}^{3,1,4}$
`mutex`	3ms	643ms	N/A
`lofree`	16ms	16.1s	N/A

(b) Model checking for SysSpec2 with SPIN (exhaustive search)

	$\mathcal{S}_{\mathrm{SK}}^{1,1,2}$	$\mathcal{S}_{\mathrm{SK}}^{2,1,3}$	$\mathcal{S}_{\mathrm{SK}}^{3,1,4}$
`mutex`	9ms	1.05s	N/A
`lofree`	34ms	93.0s	N/A

(c) Model checking with Maude (exhaustive search)

	$\mathcal{S}_{\mathrm{SK}}^{L,1,2}$	$\mathcal{S}_{\mathrm{SK}}^{L,1,3}$	$\mathcal{S}_{\mathrm{SK}}^{L,1,4}$
`mutex`	8ms	2.21s	N/A
`lofree`	340ms	54.1s	N/A

(d) Model checking `mutex` for the modified SysSpec1 with SPIN (bitstate hashing)

	$\mathcal{S}_{\mathrm{SK}}^{1,1,2}$	$\mathcal{S}_{\mathrm{SK}}^{2,1,3}$	$\mathcal{S}_{\mathrm{SK}}^{3,1,4}$	$\mathcal{S}_{\mathrm{SK}}^{4,1,5}$
time	3ms	375ms	4.07×10^3s	1.52×10^5s
Hf	3.0×10^7	1.7×10^5	45	1.9

N/A stands for Not Available. For each cell marked with N/A, the measurement did not complete because the size of the memory used was not enough. Hf stands for Hash Factor.

- Algorithms can be written directly in PROMELA without explicitly considering transition systems modeling the algorithms.

Therefore, non-experts can use SPIN for model checking some properties for algorithms.

B. Performance

We show the measurements of how fast SPIN model check the two properties for both the modified SysSpec1 and SysSpec2. Table I (a) and (b) show the measurements. The measurements were taken on Windows XP installed on a laptop with 2.33GHz CPU and 3GB memory. SPIN model checked the two properties for the modified SysSpec1 faster than for SysSpec2. The measurements suggest that an abstract mathematical model (a transition system) of an algorithm should be made so as to use SPIN efficiently.

We also show the measurements of how fast Maude model checks the two properties for the Maude system specification of $\mathcal{S}_{\mathrm{SK}}^{L,M,N}$ in Table I (c)[3]. The measurements were taken on Red Hat Linux installed on VMware Workstation, which was assigned 1GB memory, running on the Windows XP. When Table I (a) and (c) are compared, SPIN is faster than Maude for every case if they return results. When Table I (b) and (c) are compared, SPIN and Maude show a comparable

[3]The fairness used in [1] is that any messages in the network will be eventually delivered to the destinations. The fairness works for the situation where there are two nodes, but does not for three nodes or more. In the measurements described in this paper, in addition to the fairness, we used additional fairness assumptions that each node at label l1 will eventually get to either label cs or label l5, and each node at label cs will eventually get to label rem.

performance, which is consistent with one of the conclusions in [6]. Although a definitive answer to performance cannot be drawn from this experiment only, SPIN is faster than Maude when an abstract mathematical model (a transition system) of an algorithm is made.

In every measurement described so far, exhaustive search was used. When exhaustive search is used, both SPIN and Maude quickly encounter the notorious state explosion problem and so does any other model checker. SPIN provides a non-exhaustive search method that uses bitstate hashing. Bitstate hashing typically uses a small number of bits (two bits by default) to store information that a state has been visited. Hence, states are not actually visited when their hash values are the same as those of the states that have been visited so far. Since a fixed memory is used for a hash table, however, a large reachable state space can be treated. Table I (d) shows the measurements obtained by checking the mutual exclusion property for the modified SysSpec1 with SPIN using bitstate hashing. Three bits were set for each state, namely that three hash functions were used. The bitstate hashing method was able to check the property even when the number of nodes is four or more, in which exhaustive search was not. A hash factor is a predictor function of how much a reachable state space is covered by the bitstate hashing method. The higher the hash factor is, the higher the coverage is.

We used the bistate hashing method where three hash functions were used to analyze the relation between hash factors and coverages by checking the mutual exclusion property for the modified SysSpec1 where there were three nodes. A hash factor of 1.9 approximately corresponds to a coverage of 2%, and a hash factor of 42 approximately corresponds to a coverage of 99.9%. This result is consistent with the relation between hash factors and coverages described in [7] in which two hash functions are used instead of three. When the number of nodes is five, the coverage is very low (2%). It is much better than nothing, however, that a large problem can be treated even though the hash factor is very low. Since the bitstate hashing method does not give false negatives, it may find a true error.

VIII. Conclusion

We have reported on a case study, which is to model check the mutual exclusion property and the lockout freedom property for the Suzuki-Kasami algorithm with SPIN. The lessons learned from the case study are as follows:

- Transition systems can be naturally written in PROMELA
- Queues do not have to be encoded. Channels can be used as bounded queues.
- Algorithms can be written directly in PROMELA without explicitly making transition systems of the algorithms.
- Transition systems of algorithms should be made to use SPIN efficiently.
- The built-in weak fairness is useful to model check liveness properties.
- At most one transition in a process should become executable to model check liveness properties under weak fairness whose units are processes.
- SPIN can model check properties for algorithms faster than Maude when transition systems of the algorithms are explicitly made.
- The bitstate hashing method is able to deal with large reachable state spaces that cannot be exhaustively searched.

One consequence of the lessons is as follows: SPIN is suitable for the case in which you would like to model check properties for algorithms that do not use compounded data types except for queues.

References

[1] K. Ogata and K. Futatsugi, "Comparison of Maude and SAL by conducting case studies model checking a distributed algorithm," *IEICE Trans. Fundamentals*, vol. E90-A, pp. 1690–1703, 2007.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

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

[4] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM TOCS*, vol. 3, no. 4, pp. 344–349, 1985.

[5] G. J. Holzmann, *The SPIN Model Checker – Primer and Referece Manual*. Addison Wesley, 2004.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," in *4th WRLA*, ser. ENTCS 71. Elsevier, 2004, pp. 162–187.

[7] G. J. Holzmann, "An analysis of bitstate hashing," *Formal Methods in System Design*, vol. 13, no. 3, pp. 287–307, 1998.