Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Improvement of a Library for Model Checking under Weakly Ordered Memory Model with SPIN

Kosuke Matsumoto[1,a)] Tomoharu Ugawa[1,b)] Tatsuya Abe[2,c)]

Received: July 23, 2017, Accepted: October 1, 2017

Abstract: Modern multi-core CPUs might execute memory access instructions of programs out-of-order. However, the SPIN model checker does not check out-of-order executions: it only checks in-order executions. We have developed a library for SPIN that enables checking such out-of-order executions with respect to two memory models, the total store ordering (TSO) and the partial store ordering (PSO). This library provides models of variables shared with multiple threads (shared variables), and read and write macros to access them. Nevertheless, this library has three problems. First, although SPIN accepts Linear Temporal Logic (LTL) formulas, which are used for representing properties to be checked such as safety and liveness, our library did not support LTL formulas referring to shared variables. Secondly, guard statements, which are often used for blocking threads while a guard is not executable, cannot refer to shared variables. Finally, the user was unable to specify initial values of shared variables, but they are initialized with zero. As presented herein, we improved the library to resolve these problems. We produced models using our improved library and investigated the library performance.

Keywords: SPIN, memory model, LTL, model checking

1. Introduction

The model checker SPIN is used for checking programs [7]. When we use SPIN, we make a model that models behavior of the program such as its operations on variables and conditional branches. The model is written in the Promela language, which is similar to procedural languages and has a sequential execution semantics. Then SPIN generates an automaton from the model and checks all the states exhaustively if they satisfy the property that we want to check.

In model checking, SPIN checks the executions in which instructions are executed in the order described in the program. However, modern multi-core CPUs might complete instructions following to a memory access instruction before the preceding instruction is completed if they are independent of the preceding instruction. When memory access instructions of a thread in a multi-thread program are completed in a different order from that of the program, another thread might observe as if the memory access instructions are *executed* in a different order from that of the program. Therefore, it is insufficient to check only those executions in which instructions are executed in the order described in the program. Conditions under which a CPU might complete the subsequent instruction before completion of the preceding instruction are defined in the memory consistency model (memory model) of the CPU. Memory models are different among CPUs. For model checking of a multi-thread program, it is necessary to check every execution according to the memory models. However, it is an error-prone task to produce a model that simulates all the executions that satisfy the memory model because such a model tends to be complicated.

To facilitate modeling, we developed a library that enables SPIN to check executions of multi-thread programs respecting the order of completions of reads and writes to the variables that might be accessed by multiple threads (shared variable) under memory models [13]. The library provides a model of shared variables and macros for reading from and writing to the shared variables from the model that the user makes. Using this library, the user can produce a model to check all the possible execution orders according to the memory model with a similar effort to making a model that disregards the memory model. Macros provided by the library are used for reading from and writing to shared variables. `WRITE` is a macro for writing a value of the second argument to the shared variable specified by the first argument. For example, `WRITE(x, 1)` writes one to shared variable `x`. `READ` reads a value of the shared variable specified by the first argument and stores it to the local variable specified by the second argument. For example, `READ(x, r)` stores the value of the shared variable `x` to the local variable `r`.

The library has three problems. First, linear temporal logic (LTL) formulas denoting properties to be checked cannot refer to shared variables. LTL formulas are useful to express time related properties such as safety and liveness of a program. SPIN allows the users to express the properties with LTL formulas that refer to variables in the model and the point of execution, i.e., the program counters, of each process. For example, the LTL formula

```
[] (want0 == 1 -> <> t0@CS)
```

denotes the property by which it always holds that, if shared vari-

[1] Kochi University of Technology, Kami, Kochi 782–0003, Japan
[2] Software Technology and Artificial Intelligence Research Laboratory, Chiba Institute of Technology, Narashino, Chiba 275–0016, Japan
[a)] matsumoto@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp
[c)] abet@stair.center

able want0 becomes one, then process t0 eventually reaches label CS. This is the liveness property for the model of the Peterson's mutual exclusion algorithm presented in Section 2. However, the users cannot use the above LTL formula with the library because the model must read a value of shared variable to a local variable using READ to use the value.

Secondly, guard statements cannot refer to shared variables. Promela has a construct called a guard statement to block thread execution until a condition is satisfied. Using guards simplifies models and reduces the number of states to be explored by model checking. For example, the guard used in Fig. 2 in Section 2

```
(turn == 0 || want1 == 0)
```

briefly expresses the behavior by which the thread is blocked until the property of turn == 0 || want1 == 0 is satisfied. However, the guard statement above cannot be used together with the library because of the same reason as the first problem. The model must read a value of shared variable to a local variable by using READ to use the value. Therefore, the guard above must be rewritten to a loop polling the shared variable as followings:

```
do
::true -> atomic {
            READ(turn, x);
            READ(want1, y);
            if
            ::(x == 0 || y == 0) -> break;
            ::else -> skip;
            fi;
          }
od;
```

Such a rewriting is sometimes difficult if guards are used as conditions of non-deterministic choices.

Thirdly, the user cannot initialize a shared variable with a specific value. WRITE is the only way to write to the shared variables provided by the library. However, it can only be used in user-defined processes (see Section 4 for details). Therefore, no way exists to initialize the shared variables before invoking the first user-defined process. Shared variables are initialized with zero according to the specification of Promela.

In this research, we improved the library to solve these problems [*1]. The improved library is superior to existing model checkers that check executions according to memory models [3], [15], [16], [17] in the following respects: Models with this library have high readability, the properties can be expressed with LTL, and the models with this library are lightweight.

In the remainder of this paper, we first explain the SPIN model checker (Section 2), the memory model (Section 3), and the library before improvement (Section 4) as preliminaries. Subsequently, we improve the library to solve the problem described in this section (Section 5), implement the improved library, and evaluate the performance of the improved library (Section 6). Additionally, we refer to related work of model checking that deals with memory models and LTL formulas (Section 7). Finally, we discuss a summary of this paper and future work (Section 8).

[*1] This library and the model used for this research are published at https://github.com/plasklab/mmlib.

2. SPIN

The SPIN model checker is an automatic verification tool that checks hardware, software and protocol based on the model checking method [7].

2.1 Model Checking

Model checking is a formal method that exhaustively checks the states of the model devoting attention to behavior in which the state of checking target changes [6]. For example, when we check a program, we produce a model that models behavior of the program such as its operations on variables and conditional branches. We check all the states that are values of variables that can be taken when executed on the computer, the execution position of the program and so on exhaustively. A model to check using SPIN is made by describing the program and the property we want to check in Promela. After producing such a model, then the SPIN checks exhaustively if the property that we want to check is satisfied at all the states. Generally, the model checking has a defect by which they can handle only targets of checking that fit in a finite state. For example, SPIN cannot check executions for arbitrary natural numbers. Furthermore, a defect exists by which the number of states to be checked increases exponentially as the model expands. Consequently, when producing a model, the ingenuity to suppress the number of states is important.

2.2 Promela

Promela has syntax like C, and we can use the macros of C. It is suitable for model description of multi-thread program and has a construct to describe process corresponding to the thread. Statements written in each process are interleaved with the statements of other processes and are executed in the described order. In Promela, an expression delimited by "**;**" or "**->**" is treated as a statement. Every statement in the model is evaluated to true or false. False statements are not executed. This process is designated as *blocking*, which means that the execution of the process stopped by a false statement is not executed.

As an example, we model a program in C of the Peterson's mutual exclusion algorithm shown in **Fig. 1** with Promela. First, we explain the program of Fig. 1. The global variables want0 and want1 in Fig. 1 are the flags. Each flag informs the other thread that the thread t0 and t1 are about to enter the critical section. The global variable turn expresses the thread that can preferentially enter the critical section at that point. First, for t0 to enter the critical section, t0 notifies the other thread that t0 is about to enter the critical section by setting want0 to one. Next, t0 manipulates turn so that the other thread preferentially enters the critical section. Then, t0 waits until want1 becomes zero or turn becomes zero by manipulation of the other thread. If conditions are satisfied, t0 exits the wait state and enters the critical section. Finally, when t0 leaves the critical section, t0 informs the other thread of this fact by setting want0 to zero. **Figure 2** is a model of Fig. 1. In Promela, we describe a process corresponding to a thread by using the keyword proctype. In addition, an init process that is executed before the user-defined process is useful. The user-defined process is executed with the run instruction as

```
1  #include <pthread.h>
2
3  int want0 = 0, want1 = 0, turn = 0;
4
5  void *t0() {
6    want0 = 1;
7    turn = 1;
8    while (true)
9      if (turn == 0 || want1 == 0)
10       break;
11   /*Critical Section*/
12   want0 = 0;
13   return 0;
14 }
15
16 void *t1() {
17   want1 = 1;
18   turn = 0;
19   while (true)
20     if (turn == 1 || want0 == 0)
21       break;
22   /*Critical Section*/
23   want1 = 0;
24   return 0;
25 }
```

Fig. 1 Mutual exclusion program using Peterson's algorithm in C.

```
1  int want0 = 0, want1 = 0, turn = 0;
2
3  proctype t0() {
4    want0 = 1;
5    turn = 1;
6    (turn == 0 || want1 == 0);
7  CS: /*Critical Section*/
8    want0 = 0;
9  }
10
11 proctype t1() {
12   want1 = 1;
13   turn = 0;
14   (turn == 1 || want0 == 0);
15 CS: /*Critical Section*/
16   want1 = 0;
17 }
18
19 init {
20   atomic {
21     run t0();
22     run t1();
23   }
24 }
```

Fig. 2 Promela model for the program in Fig. 1.

in line 21 [*2]. The range enclosed by `atomic { }` in lines 20–23 is executed as a group of instructions and is not interleaved as far as it can be executed. The repetition is described between `do` and `od`. However, the wait state of Fig. 1 by `while` is modeled using the blocking statement of line 6 of Fig. 2. A statement described as a condition for executing the following statement, such as line 6, is called a guard. Although not used in Fig. 2, the conditional branch is written between `if` and `fi` as follows. The statement followed by "->" is executed if the guard after "`::`" is true.

```
if
  ::(x == 0) -> x = 1;
  ::(y == 0) -> y = 1;
fi;
```

The statement following one of the guards is executed nondeterministically if more than one guard gets true simultaneously. However, if all statements following "`::`" are false, then the process is blocked. In Promela, in addition to integer type variables, channel type variables are prepared. They can be treated as FIFO queues. Furthermore, it is possible to describe a label for use in an LTL formula or the like. For example, we can describe a label such as `CS:` as line 11.

2.3 How to Describe the Properties

SPIN has two methods to describe properties that models should satisfy. One is to use `assert` statements. The other is to use LTL formulas.

In the former method, we insert a statement `assert(` ψ `)` at an arbitrary place in a model where the formula ψ is defined as presented below:

$$\psi ::= \top \mid P \mid \neg\psi \mid \psi \vee \psi$$

where $\top$ denotes a propositional constant denoting truth. P is a propositional variable that is available in SPIN, for example, equations (==) or inequation (<) between variables and immediate values. For additional details, see Ref. [7]. A formula $\neg\psi$ denotes the negation. A formula $\psi_0 \vee \psi_1$ denotes the disjunction of ψ_0 and ψ_1. The conjunction and implication are abbreviations using them.

SPIN does nothing if a formula is satisfied at the place in a model and halts with an error otherwise [*3].

Although an `assert` statement matters whether the argument formula is satisfied at the place, an LTL formula represents a temporal property (about the present and future) such that, for a given property, the truth value of the property is eventually satisfied. LTL formulas φ are defined as

$$\varphi ::= \top \mid P \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathrm{X}\varphi \mid \varphi \,\mathrm{U}\, \varphi \ .$$

A formula $\mathrm{X}\varphi$ denotes that φ is satisfied at the next state. A formula $\varphi_0 \,\mathrm{U}\, \varphi_1$ denotes that φ_0 is satisfied until φ_1 is satisfied, and φ_1 is eventually satisfied. We define the following notation, for convenience:

$$\Diamond\varphi \equiv \top \,\mathrm{U}\, \varphi$$
$$\Box\varphi \equiv \neg\Diamond\neg\varphi \ .$$

Formulas $\Diamond\varphi$ and $\Box\varphi$ denote that φ is eventually satisfied and that φ is always satisfied.

We define semantics for LTL formulas. A set of execution paths that SPIN explores can be regarded as a transition system. We define the transition system as a Kripke model $\mathfrak{M} = \langle M, R, val\rangle$, i.e., M is a set of states, R is a relation on M, val is a function from the set of propositional variables to the set of subsets of M. We define a *path* π on $\mathfrak{M}$ as a (finite or infinite) sequence of M, where any adjacent pair is related by R. A state $\pi(i)$ denotes the i-th state of π where $\pi(0)$ is the first state of π. We write π^i as the suffix of π starting at i. We define satisfiability of an LTL formula φ by a path π on $\mathfrak{M}$ as

$$\mathfrak{M}, \pi \vDash \top \iff \text{true}$$
$$\mathfrak{M}, \pi \vDash P \iff \pi(0) \in val(P)$$
$$\mathfrak{M}, \pi \vDash \neg\varphi \iff \mathfrak{M}, \pi \nvDash \varphi$$

[*2] We can also describe a process that starts automatically with the keyword `active proctype`, but it is not addressed in this research.

[*3] Although we can run SPIN without being stopped even if an error is detected, we do not address that case in this paper.

$\mathfrak{M}, \pi \vDash \varphi_0 \vee \varphi_1 \Longleftrightarrow \mathfrak{M}, \pi \vDash \varphi_0$ or $\mathfrak{M}, \pi \vDash \varphi_1$

$\mathfrak{M}, \pi \vDash \mathrm{X}\,\varphi \Longleftrightarrow \mathfrak{M}, \pi^1 \vDash \varphi$

$\mathfrak{M}, \pi \vDash \varphi_0 \,\mathrm{U}\, \varphi_1 \Longleftrightarrow \mathfrak{M}, \pi^j \vDash \varphi_1$ for some $j \in \mathbb{N}$,
and $\mathfrak{M}, \pi^i \vDash \varphi_0$ for any $i < j$.

By definition,

$\mathfrak{M}, \pi \vDash \Diamond \varphi \Longleftrightarrow \mathfrak{M}, \pi^i \vDash \varphi$ for some $i \in \mathbb{N}$

$\mathfrak{M}, \pi \vDash \Box \varphi \Longleftrightarrow \mathfrak{M}, \pi^i \vDash \varphi$ for any $i \in \mathbb{N}$

hold. We write $\mathfrak{M}, m_0 \vDash \varphi$ if $\mathfrak{M}, \pi \vDash \varphi$ is satisfied for any path π with a starting state is $m_0 \in M$ on $\mathfrak{M}$.

In Promela, the implication, $\Diamond$, and $\Box$ are denoted by `->`, `<>`, and `[]`.

In Fig. 2, a liveness property is shown by which if `want0 == 1` is satisfied, then it always holds that the line at `t0@CS` is eventually processed. That is denoted as

```
[](want0 == 1 -> <> t0@CS) .
```

Given an LTL formula that represent a property to be verified, SPIN translates the negation of the LTL formula into a special process called a never process [*4]. A never process runs independently of other processes. If the negation of the LTL formula is satisfied, then the Büchi automaton corresponding to the never process entries a loop including acceptance states, and SPIN halts with an error.

3. Memory Model

Modern multi-core CPUs might complete instructions following a memory access instruction before the preceding instruction completes if they are independent of the preceding instruction. As a result, each thread might observe a sequence of memory access instruction executed by another thread as if the thread executed the instructions in a different order from the program in multi-thread programs. The conditions under which a CPU might complete the subsequent instruction before completion of the preceding instruction are defined in the memory model of the CPU. Memory models are different from CPU to CPU.

In this research, we consider write instructions (Stores) and read instructions (Loads). Here, we write $A \rightarrow B$ when a thread executes memory access instructions A and B in this order. The four possible execution orders of two memory access instructions are Store→Store, Store→Load, Load→Store, and Load→Load. In this research, we regard a memory model as a definition specifying whether a CPU might complete the subsequent instruction before completion of the preceding instruction for each of the above four execution orders when its two memory access instructions access different memory addresses. When a sequence of memory access instructions executed by a thread might be observed by another thread as if the instructions were executed in a different order, we say that *the execution order is relaxed*.

An architecture that has a memory model with some relaxed execution orders might yield different results than those a user expected. Such an architecture provides a fence instruction that forces an execution order of memory access instructions. By inserting the fence instruction, the memory access instructions preceding the fence are always complete before the subsequent memory access instructions to the fence.

[*4] Another method is to insert the LTL formula in a Promela model.

```
Initial value x = 0; y = 0;
   Thread A          Thread B
Store(x, 1);      Store(y, 1);
Load(y, eax);     Load(x, ebx);
```

Fig. 3 Example of program that can yield different results in SC and TSO.

In this research, we discuss three memory models: sequential consistency (SC) [10], total store ordering (TSO) [4], and partial store ordering (PSO) [4]. In SC, any execution order is not relaxed. In TSO, only execution order Store→Load is relaxed. In PSO, Store→Store is relaxed as is Store→Load. **Figure 3** presents an example of a program that can yield different results when executed under SC and TSO. In Fig. 3, `Store(x, 1)` writes one to memory address `x`. `Load(y, eax)` reads the value from memory address `y` and stores it in the `eax` register. For example, if thread A reads zero from `y` on the second line, then thread B always reads one from `x` on the second line in SC. In TSO, however, thread B might read zero because execution order Store→Load is relaxed.

4. Previous Version of Library

In this section, we explain the previous version of the library. We improve this library in Section 5.

The library provides a model of shared variables and macros for reading from and writing to the shared variables according to memory models. The models that the users write are intended to access shared variables by using these macros. It happens only through the shared variables that a thread observes the effect of the memory model. It observes that the CPU completes memory access instructions in a different order from that of the program. Consequently, a model incorporating the memory model merely requires properly modeled shared variables and methods of accessing it. For local variables, which are unaffected by the memory model, the models that the users write are intended to access them in the standard manner in Promela. In the remainder of this paper, we refer to a model written by the user as a *user model*. We also refer to a process defined by the user except for `init` in user models as a *user process*.

The library deals with SC, TSO, and PSO. Under these memory models, only the execution orders of memory access instructions Store→Load and Store→Store can be relaxed. Both of these execution orders allow succeeding memory access instructions to be completed before the preceding Store. Consequently, to model memory access instructions according to these memory models, some mechanism is necessary that might defer reflection of the result of a Store instruction to memory until the results of the subsequent memory access instruction are reflected. The library realizes the behavior according to TSO and PSO using the *store buffers* and the *memory process*. The library uses the *store buffer* to hold the information of Stores executed by a user process. The *memory process* is a special process to reflect the contents of the store buffer to memory. We must also consider that the instructions that a thread observes as if they were executed in a different order from the program are those that are executed by other threads. In other words, a thread can load the value stored by

itself immediately. To realize this, the library uses the *copy of shared memory* for each thread that records the information of Stores executed by the thread. When a thread reads a variable written by the Store executed by the thread, and when the Store has not been reflected to shared memory, it reads from the copy of shared memory rather than shared memory.

In this design, the memory process cannot function while a user process is executing an atomic block. A case in which the store buffer becomes full and the user process blocks is an exception. Therefore Stores are not reflected to shared memory until the process leaves the atomic block. However, the execution where Stores in the store buffer are reflected to the shared memory during the execution of the atomic block would yield the same result as the execution where those Stores are reflected immediately after the atomic block because 1) the user process executing the atomic block can load the Stores executed by itself without referring shared memory, and 2) other user processes cannot run.

The users of the library must give parameters, such as the numbers of user processes and shared variables appear in the user model, to the library. More specifically, the users must define the following three parameters as macros before including the library in the user model: the number of user processes (`PROCSIZE`), the number of shared variables (`VARSIZE`) and the size of the store buffers (`BUFFSIZE`). We must be aware of the risk that we cannot check some cases because `BUFFSIZE` works as a limitation on the number of Stores that can be relaxed simultaneously. Giving correct parameters is responsible to the library users.

The macros provided by the library are listed below. The macros expect the *memory address* instead of a shared variable because shared variables are modeled as memory addresses, which are integers between zero and `VARSIZE` − 1.

`WRITE(`*a*`, `*v*`)`**
: performs a Store according to the memory model. *a* is the memory address to which the value *v* is written.

`READ(`*a*`, `*x*`)`**
: performs a Load according to the memory model. *a* is the memory address from which a value is read, and the value is stored to the local variable *x*.

`FENCE()`
: performs a fence operation.

Real CPUs provide atomic memory access instructions such as a compare-and-swap (CAS) instructions in addition to these three instructions. Although this library does not provide macros corresponding to such instructions, some of them can be realized by combining the provided macros. For example, the CAS instruction of SPARC TSO can be realized by a combination of `READ`, `WRITE` and `FENCE` macros in an atomic block of Promela.

This library is implemented as explained below.

The library implements the set of shared variables as an array for every memory models.

For SC, `WRITE` writes to a shared variable and `READ` reads from it in the same manner as a standard array access in Promela. `FENCE` does nothing.

In TSO, `WRITE` can be reflected to memory after the subsequent `READ` because execution order Store→Load is relaxed. Consequently, the library uses the memory process, the store buffers, which are managed in a FIFO manner, the copies of shared memory, and the counters to count the Stores for each shared variable stored in the store buffer. A store buffer, a copy of shared memory, and counters are provided for each user process. They are arranged in arrays that are indexed by the process ID of each user process. The memory process executes the following steps atomically to reflect the Stores of each user process to the shared variables.

(1) The memory process selects a user process nondeterministically and fetches the least recent Store from it.
(2) The memory process reflects the Store fetched in step (1) to the shared variable.
(3) The memory process decrements the counter that belongs to the user process selected in step (1) and which corresponds to the shared variable to which the Store is reflected in step (2).

`WRITE` executes the followings atomically as shown in **Fig. 4**.

- `WRITE` inserts a pair of the memory address and the value to be written, both of which are given as its parameters, to the store buffer (`queue`) of the user process that is executing `WRITE`.
- `WRITE` writes the value to be written to the copy of the shared variable (`buffer`) corresponding to the memory address.
- `WRITE` increments the counter (`counter`) corresponding to the memory address.

In Fig. 4, `_pid` is the process ID that is executing `WRITE`, `s` is the memory address, and `v` is the value to be written. As shown in **Fig. 5**, `READ` atomically

- reads the value from either the shared variable (`shared_memory`) indexed by the memory address given as a parameter or the copy of it, depending on the contents of the store buffer of the user process that is executing `READ`, and
- writes it to the local variable specified by the parameter.

Because the memory process reflects the contents of the store buffer to the shared variable while interleaving with the user process, the preceding Store can be reflected to the shared variable after the subsequent Load is executed as a result of interleaving. Therefore, Store→Load is relaxed. `FENCE` immediately reflects

```
1  #define WRITE(s, v) {\
2  atomic {\
3    queue[_pid]!s,v;\
4    buffer[_pid * VARSIZE + (s)] = v;\
5    counter[_pid * VARSIZE + (s)]++;\
6  }\
7  }
```

Fig. 4 Implementation of `WRITE` macro for TSO.

```
1  #define READ(s, v){\
2  atomic {\
3    if\
4      ::(counter[_pid * VARSIZE + (s)] == 0)\
5        -> v = shared_memory[s];\
6      ::else\
7        -> v = buffer[_pid * VARSIZE + (s)];\
8    fi;\
9  }\
10 }
```

Fig. 5 Implementation of `READ` macro for TSO.

```
1 #define WRITE(s, v){\
2 atomic {\
3   (len(queue[(_pid)*VARSIZE+(s)]) < BUFFSIZE);\
4   queue[(_pid) * VARSIZE + (s)]!v;\
5   buffer[(_pid) * VARSIZE + (s)] = v;\
6   gcounter++;\
7 }\
8 }
```

Fig. 6 Implementation of WRITE macro for PSO.

```
1 #define READ(s, v){\
2 atomic {\
3   if\
4     ::(len(queue[(_pid)*VARSIZE+(s)]) == 0)\
5       -> v = shared_memory[s];\
6     ::else -> v = buffer[(_pid)*VARSIZE+(s)];\
7   fi;\
8 }\
9 }
```

Fig. 7 Implementation of READ macro for PSO.

all of the store buffer contents to the shared variable.

In PSO, execution order Store→Store is relaxed: WRITE might be reflected to memory after the subsequent memory access instruction. Consequently, a store buffer is provided for each user process and each shared variable. WRITE inserts only the value to be written to the store buffer corresponding to the target shared variable among the store buffers corresponding to the process ID of the user process, as presented in **Fig. 6**. In Fig. 6, gcounter is a counter that counts the total number of Stores held in the store buffer. For READ, unlike TSO in which READ refers to the counter to count the Stores stored in the store buffer, the len function is used to obtain the length of the channel. Consequently, PSO does not need the counters as shown in **Fig. 7**. FENCE is the same as TSO. Because separate store buffers are provided for each shared variable, the preceding Store may be reflected to the shared variable after a subsequent Store is reflected by interleaving if these two Stores write to different shared variables. Therefore, Store→Store is relaxed. Store→Load is relaxed for the same reason as TSO.

In the remainder of this section, we describe the use of macros provided by the library.

Figure 8 is a similar model to that of Fig. 2 in which global variables and processes t0 are modified to use the library. Lines from 1–3 define macros for the parameters. Because two processes exist, t0 and t1, PROCSIZE is defined as two. Because three shared variables are used, want0, want1 and turn, VARSIZE is defined as three. BUFFSIZE is defined as three because the number of writes to the same shared variable in any process is three at most. The following lines, lines 4–6, are macros for improving the model readability. These macro definitions allow the user to address shared variables as intuitively as the user model without this library because shared variables are modeled as integers from zero to VARSIZE – 1. The file tso.h included in line 7 is the library. This file is for checking executions according to TSO. From the line 8 onward is the model in which writing and reading accessing shared variables are modified to use macros provided by the library. Lines 8–10 of Fig. 2 were rewritten to a model that includes repetition as lines 14–24 of Fig. 8, because we cannot describe guards referring to shared variables with this

```
1  #define PROCSIZE 2
2  #define VARSIZE 3
3  #define BUFFSIZE 3
4  #define want0 0
5  #define want1 1
6  #define turn 2
7  #include "tso.h"
8
9  proctype t0() {
10   int eax;
11   int ebx;
12   WRITE(want0, 1);
13   WRITE(turn, 1);
14   do
15     ::true -> atomic {
16                 READ(turn, eax);
17                 READ(want1, ebx);
18                 if
19                   ::(eax == 0 || ebx == 0)
20                     -> break;
21                   ::else -> skip;
22                 fi;
23               }
24   od;
25 CS: /*Critical Section*/
26   WRITE(want0, 0);
27 }
```

Fig. 8 Part of Promela model with the library of the previous version for a mutual exclusion program using Peterson's algorithm.

library in the same manner as we can do in Promela without this library.

5. Improve Library

The library has two restrictions. First, the model must read a value of a shared variable to a local variable using READ to use the value. For example, the user cannot refer directly to a value of a shared variable in expressions. Secondly, the user can use WRITE and READ only in the user process, because both are implemented such that they are dependent on the process ID of the executed process. For example, it cannot be used in an init process or in an LTL formula. In this section, we extend the library to provide new macros that can solve problems arising from these restrictions.

When the user writes an expression referring to the value of a shared variable, it is necessary to store the shared variable in a local variable and evaluates the expression incorporating the local variable. However, because each process is executed while interleaving, even if the value of the shared variable is read immediately before evaluating the expression, it is not necessarily the latest value when evaluating the expression. Therefore, the user cannot use the guard referring to the shared variable in the user model as it is when using the library. It is necessary to rewrite the user model as shown in lines 14–24 of Fig. 8. Therefore, for this research, we provide a new READ, which directly returns a value of a shared variable instead of the previous READ, which stores a value of a shared variable in the local variable. The new READ macro makes it possible to refer the shared variable from a guard because of this. In this research, we provide a new READ macro as explained below.

READ(*s*)

performs a Load according to the memory model and returns the value of the shared variable *s*.

Because shared variables are managed by this library, the user needs to use WRITE to write a value to a shared variable. How-

ever, WRITE is useful only in the user process although an initial value of a shared variable must be set before executing the user process. As a result, the initial value of the shared variable becomes zero according to the specification of Promela. Therefore, this research provides a macro for setting initial values as follows, so that shared variables can be initialized in the init process.

INIT(*s*, *v*)

sets the initial value of the shared variable *s* to *v*.

This macro writes directly to the array of shared variables without going through the store buffer.

When performing a check using an LTL formula, the LTL formula cannot refer to shared variables because READ of the previous version cannot refer directly to shared variables. Furthermore, an LTL formula that refers to a shared variable cannot be described even if a new READ is used because READ depends on the process ID of the user process that executes READ. Therefore, this research makes it possible to refer shared variables from an LTL formula by providing a dedicated macro GSVAR. However, the value of memory observable for each process might be different depending on the memory model. Consequently, in this research, in addition to the GSVAR macro, which refers to a value of a shared variable reflected in the shared memory, we also provide the SVAR macro, which refers to a value of a shared variable observable from a certain process, as follows.

GSVAR(*s*)

returns the value of the shared variable *s*.

SVAR(*p*, *s*)

returns the value of the shared variable *s* that are observed from the user process *p*.

If we prepare a process that does not write to any shared variables at all, then the result of SVAR executed by that process will always be the same as GSVAR. However, this research provides a mechanism to refer to the value of each shared variable as a GSVAR macro, because the process prepared for the realization of GSVAR is a useless process, and is irrelevant to the user model.

Below, we describe the implementation and usage of this newly provided macro.

5.1 Guard Macro

In this section, we describe the implementation and usage of a new READ, which directly returns the value of the shared variable to refer to the shared variable from the guard.

SC requires no special ingenuity. The new READ is implemented in the way that a value of the shared variable of the argument is returned in the normal manner. For TSO, we implemented READ as follows. If a Store targeting the same shared variable as the argument of READ does not exist in the store buffer corresponding to the process ID of the process that executed READ, then READ returns the value of the shared variable; otherwise READ returns the value of the copy of shared memory corresponding to the process ID of the executed process, as shown in **Fig. 9**. For PSO, we implemented READ as follows. If the store buffer corresponding to the same shared variable as the argument of READ among the store buffers corresponding to the process ID of the process that executed READ is empty, then READ returns the value of the shared variable; otherwise READ returns the value of the copy of shared memory corresponding to the process ID of the executed process.

```
1 #define READ(s)\
2   (counter[_pid * VARSIZE + (s)] == 0 ->\
3       shared_memory[s] :\
4       buffer[_pid * VARSIZE + (s)])
```

Fig. 9 Implementation of new READ macro for TSO.

```
1  #define WRITE(s, v){\
2  atomic {\
3    int tmp = v;\
4    queue[_pid]!s,tmp;\
5    buffer[_pid * VARSIZE + (s)] = tmp;\
6    counter[_pid * VARSIZE + (s)]++;\
7    tmp = 0;\ /* not to create
8                 an unnecessary state */
9  }\
10 }
```

Fig. 10 Implementation of new WRITE macro for TSO.

With previous version library, the user uses READ to read the value of shared variable x to local variable a as shown below.

```
READ(x, a);
```

The new READ is used as shown below.

```
a = READ(x);
```

Using the new READ, the user can model the wait state including the repetition as lines 14–24 of Fig. 8 with one statement as follows, so that it does not include repetition.

```
(READ(turn) == 0 || READ(want1) == 0);
```

The new READ might be used in the second argument of WRITE to return the value of the shared variable directly. A problem arises when READ that reads the variable to which WRITE is writing is used as the second argument of WRITE because the location to be read with READ changes between when storing the read value in the store buffer and when storing it in a copy of shared memory in the implementation of WRITE for TSO shown in Fig. 4. Therefore, we changed the implementation of WRITE to store the second argument in the local variable at the beginning as shown in **Fig. 10**. We changed PSO in the same way.

5.2 Macros to Refer to Shared Variables from LTL Formula

In this section, we describe the implementation and usage of GSVAR and SVAR to refer to the value of the shared variable from an LTL formula.

5.2.1 Implementation

SC requires no special ingenuity. The GSVAR and SVAR are implemented in the way that a value of the shared variable of the argument is returned in the normal manner. Specifically, GSVAR is implemented to return the value of the shared variable of the argument. Also, SVAR is implemented to return the value of the shared variable specified as the second argument, irrespective of the process specified by the first argument.

For TSO, it is noteworthy that the contents written by each process are accumulated in the store buffer for each process. GSVAR is implemented to return the value of the shared variable of the argument, irrespective of the store buffer counters. SVAR is implemented such that if the Store targeting the same shared variable as the second argument does not exist in the store buffer corresponding to the process ID specified by the first argument, it

```
1 #define SVAR(p, s)\
2   (counter[p:_pid * VARSIZE + (s)] == 0 ->\
3      shared_memory[s] :\
4      buffer[p:_pid * VARSIZE + (s)])\
```

Fig. 11 Implementation of SVAR macro for TSO.

returns the value of the shared variable. Otherwise, returns the value of the copy of shared memory of the process specified by the first argument, as shown in **Fig. 11**. It is worth noting that we can refer to the process ID of a specific process by an expression `ProcessName:_pid` in Promela.

PSO is also defined in the same way as TSO.

5.2.2 Usage

When referring to the value of a shared variable in an LTL formula, it is necessary to devote attention to whether the referred value is the value held by the shared variable or the value of the shared variable observed from a process. These might be different in TSO and PSO. For example, the user wants to check that whenever the shared variable `want0` becomes one, it eventually returns to zero and the block of the following statement of process `t1` in line 14 in the model of Fig. 2 is released.

```
(turn == 1 || want0 == 0);
```

In that case, because it is important how the value of the shared variable is observed from the process `t1`, the user should use SVAR as shown below.

```
[](SVAR(t1, want0) == 1 ->
        <>SVAR(t1, want0) == 0)
```

If there is a process that does not appear in the model of Fig. 2, and the user is interested in the change in the value of the shared variable `want0` observed from that process, the user should use GSVAR to check the value of the shared variable directly as the following example.

```
[](GSVAR(want0) == 1 ->
        <>GSVAR(want0) == 0)
```

Incidentally, the initial value of the variable becomes zero when not specifying the initial value in Promela. In this research, we provide `INIT` to set the initial value of the shared variable, but the value of the shared variable remains zero until the user sets the initial value with `INIT`. Therefore, for example, an LTL formula such as `<>(GSVAR(x) == 0)` is satisfied in the initial state. Consequently, the user cannot check it correctly. However, if the user changes the user model and the LTL formula so that the initialization is completed explicitly, then such a property can also be checked. For example, a method of introducing a flag exists, indicating that model initialization has been completed as a normal global variable. The flag is manipulated at the position where initialization of the shared variable in the init process is completed. In this method, the user introduces the global variable `flag` as a flag and manipulates the flag as follows.

```
flag = 1; flag = 0;
```

Let m_0 be the state where the `flag` is set after initialization. Such a state is only one state. In addition, transform the LTL formula φ as shown below.

$$\Box(\texttt{flag} \to \varphi)$$

We prove that the original LTL formula φ and its translation $\Box(\texttt{flag} \to \varphi)$ are equivalent in a sense.

Let $\mathfrak{M} = \langle M, R, val\rangle$ and $\mathfrak{M}' = \langle M', R', val'\rangle$ be Kripke models where $m_0 \in M$ and $val(\texttt{flag}) = \emptyset$, and

$$m_0' \notin M \qquad M' = M \cup \{m_0'\}$$
$$R' = R \cup \{\langle m_0', m_0\rangle\} \qquad val'(\texttt{flag}) = \{m_0\}$$
$$val'(p) \setminus \{m_0'\} = val(p) \text{ for any } p \neq \texttt{flag}\ .$$

Lemma 1. *Assume that φ does not contain* `flag`*, then $\mathfrak{M}, \pi \vDash \varphi$ is equivalent to $\mathfrak{M}', \pi \vDash \varphi$.*

Proof. By induction on φ. □

Proposition 2. *Assume that φ does not contain* `flag`*. Then, $\mathfrak{M}, m_0 \vDash \varphi$ is equivalent to $\mathfrak{M}', m_0' \vDash \Box(\texttt{flag} \to \varphi)$.*

Proof. Assume that $\mathfrak{M}, m_0 \vDash \varphi$. Let π be a path for which the starting state is m_0' on $\mathfrak{M}'$. Because π has the prefix $m_0' m_0$, π^1 is a path with starting state m_0 on $\mathfrak{M}$. Therefore, $\mathfrak{M}, \pi^1 \vDash \varphi$ holds.

By definition,

$$\begin{aligned}
&\mathfrak{M}', \pi \vDash \Box(\texttt{flag} \to \varphi)\\
&\iff \mathfrak{M}', \pi^i \vDash \texttt{flag} \to \varphi \text{ for any } i \in \mathbb{N}\\
&\iff \mathfrak{M}', \pi^1 \vDash \texttt{flag} \to \varphi \text{ and } \mathfrak{M}', \pi^i \vDash \texttt{flag} \to \varphi \text{ for any } i \neq 1\\
&\iff \mathfrak{M}', \pi^1 \vDash \varphi\ .
\end{aligned}$$

Because $\mathfrak{M}', \pi^1 \vDash \varphi$ is equivalent to $\mathfrak{M}, \pi^1 \vDash \varphi$ by Lemma 1, $\mathfrak{M}', m_0' \vDash \Box(\texttt{flag} \to \varphi)$ holds.

Letting $\mathfrak{M}, m_0 \nvDash \varphi$, by definition, there exists a path π for which the starting state is m_0 on $\mathfrak{M}$ such that $\mathfrak{M}, \pi \nvDash \varphi$. By Lemma 1, $\mathfrak{M}', \pi \nvDash \varphi$ holds. Because $m_0 \in val'(\texttt{flag})$, $\mathfrak{M}', \pi \nvDash \texttt{flag} \to \varphi$ holds. Because the super-sequence of π with the prefix m_0' (denoted by $m_0'\pi$) is a path on $\mathfrak{M}'$, $\mathfrak{M}', m_0'\pi \nvDash \Box(\texttt{flag} \to \varphi)$ holds. Therefore, $\mathfrak{M}', m_0' \nvDash \Box(\texttt{flag} \to \varphi)$ holds. □

Those not managed by the library such as global variables other than shared variables, local variables, and labels are referred to in the LTL formula in the same way as plain Promela, as described in Section 2.3.

5.3 Validity of Modeling

Our library uses the store buffers and copies of the shared variables to model TSO and PSO. This is a standard approach to produce an operational models of memory models. In fact, earlier work often uses a similar modeling to explain memory models [12], [16]. In this section, we explain that the modeling of our library is valid by presenting correspondence to the modeling explained by Travkin et al. [16].

Travkin et al. explained the TSO and PSO memory models as models in which each process has one or more store buffers. *Their* store buffer behaves as a cache as well as a FIFO queue to defer reflecting writes. Unlike real caches, however, the cache is not guaranteed to be consistent with shared memory and store buffers of other processes. Writes performed by using a process (*write* operations) are cached in the store buffer. The process occasionally flushes the store buffer (*flush* operation) to reflect the writes. In their semantics, a single *flush* operation reflects only the oldest write to shared memory. A Read performed by using a process (*read* operation) reads from shared memory only when the value of the variable from which the process is reading is not cached in

the store buffer. The *fence* operation forces the *flush* operations to empty the store buffer.

Our library realizes two roles of their store buffer with separate mechanisms: a role as a FIFO queue to defer reflecting writes and a role as a cache. The former is realized with Promela's channels (we call these channels the store buffers) and a memory process. The latter is realized with the copies of shared variables. Another difference is that the memory process reflects writes in our library while user processes perform the *flush* operations nondeterministically by themselves to reflect writes in the model of Travkin et al. Nevertheless, they are fundamentally the same because the operation to reflect a single write is an atomic one. For that reason the result of model checking is unaffected by who performs it.

The *write* and *fence* operations of Travkin et al. correspond respectively to our `WRITE` and `FENCE` macros. The semantics of *read* operation of Travkin et al. is the same as the `READ` macro in the previous library. The *read* operation copies the value of a shared variable to a local variable given as a parameter. Strictly speaking, the *read* operation does not correspond to the `READ` macro of the new library. However, the new `READ` can emulate the *read* operation as the following example, which reads shared variable x to local variable a.

a = `READ(`x`);`

As long as the users use the `READ` macro in this manner, our library behaves under the memory model of TSO and PSO that Travkin et al. modeled.

Even when the users use the `READ` macro in a different manner, the behavior of our library does not deviate from the TSO and PSO of Travkin et al. For example, the statement

`WRITE(x, READ(y))`,

where the `READ` macro is used as a parameter for a `WRITE`, the macro behaves as the same as

`atomic{tmp = READ(y); WRITE(x, tmp)}`

because the `WRITE` macro saves the result of the `READ` macro into a temporary local variable inside it. It then uses the temporary variable in the write operation as we described in Section 5.1. Therefore, the behavior of our library does not deviate from the TSO and PSO described by Travkin et al. For the case in which the users use the `READ` macro in a guard statement, our library also behaves within the TSO and PSO of Travkin et al. For example, a model that uses a guard

`(READ(x) == 0)`

checks those execution traces of the model that is obtainable by replacing the above guard with

`atomic{tmp = READ(x); (tmp == 0)}`

that has `READ(x)` yielding zero. From this discussion, the modeling of our library corresponds to the modeling of TSO and PSO explained by Travkin et al.

6. Experiment

We investigated whether the library was implemented correctly. We also checked whether the library was able to check models correctly using LTL formulas referring shared variables and evaluated its performance.

We investigated it in the same way as in Ref. [13]. Specifically, we investigated using the test program collection (x86-TSO litmus test) [14] to ascertain whether it conforms to the memory model of x86-TSO. The x86-TSO litmus test contains the programs, possible results, and unexpected results when executing those programs under x86-TSO. We investigated whether the TSO model of the library matches the expected result of the x86-TSO litmus test. We also investigated correctness of the PSO model of the library by comparing the results with the execution results of McSPIN [3], which is a model checker considering memory models. As a result, we obtained the expected result in all tests.

In the following sections, we discuss the results of the investigation of whether the model checking using the LTL formulas can be performed correctly. Then we present the performance of the library.

6.1 Methodology

To investigate whether the user can perform checks correctly using LTL formulas and to evaluate the library performance, we used some of the models described in Refs. [5] and [11]. From Ref. [5], we used models that uses LTL formulas and whose results are stated, as shown in **Table 1**. However, because the library handles only integer shared variables, we excluded one model that uses channel-type shared variables. The "Model name" column represents the place described in Ref. [5]. Of the models presented in Table 1, Question 2.1 and 3.7 are models we used because they used LTL formulas to check liveness and safety. From Ref. [11], we used the models of mutual exclusion algorithms for two processes, as presented in **Table 2**. The "Model name" column represents the algorithm names. In Ref. [11], when checking executions according to TSO and PSO of each model, the position in which the fence instruction should be inserted for mutual exclusion to work property is indicated clearly. Therefore, the models presented in Table 2 hold safety. These models include reading of shared variables, guards that refer to shared variables, and setting initial values of shared variables.

Whether or not the macro referring to the shared variable from LTL formulas behaves correctly was judged by comparison with the checked results described in Ref. [5] and by ensuring that the property of liveness and safety hold for the model of Ref. [11].

Section 3.5.2, Question 3.1, Question 3.7 and Dijkstra are models for which initial values other than zero must be set for shared variables. The correctness of `INIT` was investigated using these models. The performance was evaluated in terms of memory usage (the number of bytes used to represent one state multiplied by the number of states) and the execution time.

The models in Table 1 and Table 2 were checked using LTL formulas. For model checking without the library of the models in Table 1, we used the original model described in Ref. [5]. When the models in Table 2 using the library, we checked the model written in plain Promela. For model checking with the library, we checked the models that we produced by modifying the models in Table 1 and Table 2 to use the library.

Following the name in the "Model name" column in Table 1 and Table 2, we described (L) if we checked liveness and (S) if

Table 1 Results of model checking for models in a textbook on model checking.

	Result							
	Document		SC		TSO		PSO	
Model name	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)
Section 3.5.2	46.5	0	129.6	0	4210.4	0.02	4991.5	0.02
Exercise 2.2	0.2	0	0.3	0.01	24.9	0.07	26.1	0.07
Question 2.1 (L)	1.7	0	5.0	0.11	130.6	0.02	1914.0	0.02
Question 2.1 (S)	2.7	0	2.5	0.02	5.3	0.13	8.8	0.13
Question 3.1	0.8	0	1.8	0	1477.4	0.01	>47 GB	>42.6
Question 3.3	1.9	0	2.3	0	150.0	0.07	146.3	0.07
Question 3.7 (L)	1.5	0	2.0	0	251.6	0.13	176.4	0.13
Question 3.7 (S)	0.8	0	1.1	0.03	36.9	0.06	25.9	0.06

Table 2 Results of model checking for correct running under the memory models.

	Result							
	Document		SC		TSO		PSO	
Model name	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)	Memory (KB)	Execution time (s)
Burns (L)	5.8	0	5.8	0	719.3	0.03	710.5	0.03
Burns (S)	2.9	0	2.9	0	369.0	0	364.5	0
Dekker (L)	17.3	0	17.3	0	1929.1	0.08	4084.2	0.08
Dekker (S)	8.9	0	8.9	0.01	981.8	0.01	2069.2	0.01
Dijkstra (L)	8.2	0	8.2	0.03	89.4	0.04	113.7	0.04
Dijkstra (S)	16.2	0	16.2	0	1365.3	0.03	2216.4	0.03
Lamport Bakery (L)	2.6	0	2.6	0	25.9	0.12	42.0	0.12
Lamport Bakery (S)	41.9	0	41.9	0	2002.6	0.01	3235.2	0.01
Lamport Fast (L)	15.5	0	15.5	0.05	396.5	0.05	237.7	0.05
Lamport Fast (S)	37.7	0	37.7	0.10	3230.2	0.03	5652.8	0.03
Peterson (L)	5.4	0	5.4	0	214.0	0.03	601.6	0.01
Peterson (S)	2.8	0	2.8	0	113.7	0	184.0	0
Peterson∗ (L)	-	-	-	-	-	-	270.9	0.03
Peterson∗ (S)	-	-	-	-	-	-	143.6	0
Szymanski (L)	9.4	0	9.4	0	1732.0	0.03	1752.2	0.03
Szymanski (S)	5.2	0	5.7	0.01	1098.7	0.02	1111.5	0.02

we checked safety. For the models in Table 1, we checked using the LTL formulas described in Ref. [5]. For the models in Table 2, we added a label `CS` representing the critical section and checked it using LTL formulas. For example, when checking the liveness of the process `P`, if `P` asserts entry of the critical section by setting the shared variable `F` to one, then the LTL formula is expressed as shown below.

```
[] ((SVAR(P, F) == 1) -> <> P@CS)
```

The liveness was checked under weak fairness with the `-f` option.

During the check process, `PROCSIZE` and `VARSIZE` within the parameters of the library are defined in accordance with the number of user processes and the number of global variables of respective models. `BUFFSIZE` is defined as five in all models because some models include user processes that perform unbounded number of iterations containing writes. Furthermore, when checking the model of Table 2 with TSO and PSO, fence instructions are as Ref. [11] suggests inserted so that mutual exclusion works properly in each memory model.

The experimental environment was Ubuntu 16.04.2 LTS, Intel Core i7-6700K 4.00 GHz.

6.2 Results

For the SC in Table 1, the results for all models matched results presented in Ref. [5]. For TSO and PSO, the results for checking liveness and safety differed from Ref. [5], but their differences can be regarded as reasonable because these models were algorithms in which mutual exclusion does not work properly in memory models where the execution order is relaxed. The model in Section 3.5.2 was unaffected by the memory model because the number of user processes is one. By this model, we confirmed that the macros for TSO and PSO worked correctly, even in models with only a single user process, because the results matched Ref. [5]. Question 3.1 was also a model that has a single user process. We confirmed that the checking result of the matched the results Ref. [5] for TSO. However, for PSO, the number of states to be explored was too large. For that reason, the checking process was terminated when 47 GB of memory was used.

For liveness in the models in Table 2, the results with SC matched the results of checking the model written in plain Promela. Safety was not satisfied when the model of Peterson was checked with PSO. We confirmed the description of Ref. [11]. The text revealed that the authors did insert the fence instruction to make the mutual exclusion work properly possible in PSO in their experiment [*5]. However, the model for PSO shown in

[*5] p.161

the document did not include fence instructions. Therefore, we checked the model which inserted the fence instruction to make the mutual exclusion work properly with PSO. We presented the result as Peterson∗. Peterson∗ confirmed that safety was satisfied with PSO and the mutual exclusion was established.

The results described above demonstrate that the checks performed using the LTL formulas referring to the shared variable were implemented correctly using the macro provided by the library.

Memory usage (KB) and execution time (s) necessary for checking each model are presented in the "Memory" and "Execution time" columns in Table 1. These results are memory usage and execution time required by SPIN to detect a counterexample or to terminate the checking without detecting a counterexample.

Every memory model for all models took more memory and execution time than the models in Ref. [5] or Ref. [11].

For example, checking the liveness of the model in Question 2.1 with PSO took 1,000 times more memory than the model in Ref. [5] which is written in plain Promela. This was true because each thread wrote to two shared variables in the model in Question 2.1, whereas each user process wrote to only one shared variable in all other models in Table 1, except for Question 3.1. In PSO, because each user process has a store buffer for each shared variable, the number of states increases greatly when checking a model in which one user process manipulates multiple shared variables. In a model where each user process writes to only one shared variable, each user process uses only a single store buffer, the number of states should be the same as that of TSO.

When checking the model of Question 3.1 with PSO, we terminated the checking process when 47 GB of memory were used because the number of states to be explored was too large. This was because, in the Question 3.1, one user process wrote to many shared variables. As described above, because PSO has the store buffer for each user process and each shared variable, in a model in which one user process writes to multiple shared variables, the number of states to be explored is larger with PSO than with TSO. Question 3.1 is a model in which one user process wrote to six shared variables. It can be expected that a huge amount of memory and execution time are necessary to check this model considering that the model of Question 2.1, which wrote to two shared variables, required 1,000 times more memory than the model written in plain Promela. In fact, such a result was actually obtained. A model with a single user process like Question 3.1 can be checked without the library because the model written in plain Promela and the model using the library are expected to give the same result in any memory model.

The discussion presented above clarifies that the library has characteristics that model checking of a model in which a single process writes to many shared variables with PSO causes a state explosion. It is worth noting the approaches that are based on the idea that uses store buffers to defer writing to shared variables [16], [17] are expected to share this characteristic.

In contrast, model checking of Question 3.7 with PSO used less memory than with TSO. In this model, the bytes necessary to express a single state were fewer with PSO than with TSO. This difference in the number of bytes arose because a counter for each process required for TSO implementation was unnecessary in PSO, and PSO inserted only a value to a store buffer, whereas TSO inserted a pair of a memory address and a value. Therefore, depending on the user model, the model size with PSO could be smaller than with TSO. Although Lamport Fast in Table 2 with PSO used less memory than with TSO, the reason was different because PSO requires more bytes to express a single state; the reason can be thought that SPIN detected that liveness did not hold with PSO earlier than with TSO because the Store→Store execution order is relaxed only in PSO.

The execution time was shorter than 0.01 s, which is the shortest time that SPIN can measure, for most models. The model that took the longest time except for Question 3.1 took 0.13 s.

7. Related Work

Before this research was undertaken, a number of methods of checking execution according to the memory model with SPIN were proposed [8], [16], [17].

Wehrheim et al. proposed a method to convert C/C++ code into Promela code that has behavior according to TSO and PSO via LLVM IR code [16], [17]. This method is based on the idea of expressing the reordering of execution order of memory access instructions that can occur in TSO and PSO by the store buffer. This respect is similar to the method used for this research. In the method explained by Wehrheim et al., the user writes the properties checked by an LTL formula and an `assert` statement as in the plain Promela. The user can refer to a value of all global variables from within an LTL formula. However, only values that are actually reflected in memory can be referenced. For that reason, the user cannot refer to values of global variables that are observable by a certain process unlike the `SVAR` instruction provided by our method. In the method used for this research, it is possible to refer not only to the values that are actually reflected in memory but also to the values of a global variable that is observable by a specific process. In addition, although Promela code generated by the method of Wehrheim et al. has poor readability, it is necessary to read the code to write the property to be checked. Using our method, execution can be checked according to TSO and PSO by virtually replacing memory access instructions for shared variables in the model without consideration the memory model to memory access instructions provided by the library. Therefore, the code readability is high, and the property to be checked can be written easily.

Wehrheim et al. use the library written in Promela, which can check execution according to SC, TSO and PSO for comparison in experiments with Refs. [16], [17]. This library is implemented based on the idea using the store buffer as with this research. However, in the library of Wehrheim et al., global variables are hidden from the user. Consequently, the user cannot write an LTL formula that refer to global variables as those of our previous library could not. The improved library overcomes the defect of the library of Wehrheim et al. by providing a dedicated macro so that the user can access shared variables (global variables that might be accessed by multiple processes).

Tomasco et al. designed an API that behaves according to the memory model and which uses CBMC [9] or Nidhugg [1] as a

backend to provide model checking according to the memory model [15]. However, the method of Tomasco et al. does not support checking using LTL formulas.

Abdulla et al. proposed a model checker that is capable of handling a memory model based on a stateless model checking [1], [2]. The model checker of Abdulla et al. only covers the multi-thread program written in C, but our method can also check multi-thread programs other than C by modeling with Promela.

8. Conclusion

As described in this paper, we improved the library that we are developing to support checking of executions according to weak memory models with SPIN. Specifically, we improved the library so that shared variables can be referred from guards. Additionally, we improved the library so that initial values of shared variables can be set. Finally, we improved the library so that shared variables can be referred from an LTL formula when the user writes the properties that the user want to check.

As a result of implementing the improved library and carrying out the experiment, we confirmed that the behavior of the macros provided by the improved library is correct. Results of evaluation of the performance demonstrated characteristics that a model in which one user process writes to many variables, causes a state explosion when it is checked with PSO. Regarding the execution time, in the experiments other than that for which the model check was aborted, the checking is completed within an execution time that is sufficiently short on a modern computer.

In future works, first of all, an investigation will be conducted to ascertain whether the behavior of TSO and PSO differs between our library and the library of Wehrheim et al. [16], [17]. In addition, supporting more memory models and cache coherency. Furthermore, it is the user's responsibility in our library to set the number of processes and the number of shared variables among the library parameters. It is urgent to develop a mechanism that extracts these parameters automatically from the program to be checked for ease of use of the library. In the library, if the write overflows from the store buffer, then the process will be blocked. But a method exists of making an error when the write overflows from the store buffer. Although one can easily to recognize that the user is not performing the check correctly if the library makes an error, the library is implemented so as to block the process considering the use in bounded model checking. Future studies should be conducted to extend the function to detect errors. Furthermore, the library does not provide instructions with different semantics for each CPU such as the CAS instruction. Some instructions can be defined by a user by combining the macros provided by the library and the functions of Promela. Nevertheless, because some instructions cannot be defined, this point must be improved so that representative instructions are definable.

Acknowledgments This research was partially supported by JSPS KAKENHI Grant Numbers 16K21335 and 16K00103.

References

[1] Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C. and Sagonas, K.F.: Stateless Model Checking for TSO and PSO, *Proc. TACAS*, LNCS, Vol.9035, pp.353–367 (2015).

[2] Abdulla, P.A., Atig, M.F., Jonsson, B. and Leonardsson, C.: Stateless Model Checking for POWER, *Proc. CAV*, LNCS, Vol.9780, pp.134–156 (2016).

[3] Abe, T. and Maeda, T.: A General Model Checking Framework for Various Memory Consistency Models, *High-Level Parallel Programming Models and Supportive Environments*, pp.332–341 (2014).

[4] Adve, S.V. and Gharachorloo, K.: Shared memory consistency models: A tutorial, *IEEE Computer*, Vol.29, No.12, pp.66–76 (1996).

[5] AIST: *Model Checking*, Kindai Kagaku Sha (2010).

[6] Clarke, E.M., Emerson, E.A. and Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM TOPLAS*, Vol.8, No.2, pp.244–263 (1986).

[7] Holzmann, G.J.: *The SPIN Model Checker*, Addison-Wesley (2003).

[8] Kato, T., Ichiba, T., Honda, S. and Takada, H.: Model Checking of the Spin-lock in Consideration of Hardware Behavior, *EMB*, Vol.2011, No.2, pp.1–8 (2011).

[9] Kroening, D. and Tautschnig, M.: CBMC - C Bounded Model Checker - (Competition Contribution), *Proc. TACAS*, LNCS, Vol.8413, pp.389–391 (2014).

[10] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE TC*, Vol.C-28, No.9, pp.690–691 (1979).

[11] Linden, A.: On the Verification of Programs on Relaxed Memory Models, Ph.D. Thesis, Universite de Liege (2013).

[12] Linden, A. and Wolper, P.: An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models, *SPIN*, LNCS, Vol.6349, pp.212–226 (2010).

[13] Matsumoto, K., Ugawa, T. and Abe, T.: A library of memory access instructions under relaxed memory models for SPIN, *FOSE*, Vol.23, pp.63–72 (2016).

[14] Owens, S., Sarkar, S. and Sewell, P.: A better x86 memory model: x86-TSO (extended version), Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory (2009).

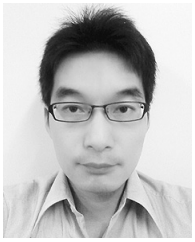
[15] Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., Torre, S.L. and Parlato, G.: Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions, *Proc. FMCAD* (2016).

[16] Travkin, O. and Wehrheim, H.: Verification of Concurrent Programs on Weak Memory Models, *Proc. ICTAC*, LNCS, Vol.9965, pp.3–24 (2016).

[17] Wehrheim, H. and Travkin, O.: TSO to SC via Symbolic Execution, *Proc. HVC*, LNCS, Vol.9434, pp.104–119 (2015).

Kosuke Matsumoto was born in 1994. He received his B.E. degree from Kochi University of Technology in 2017. He received the IEEE Computer Society Japan Chapter FOSE Young Researcher Award in 2016.

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. In 2008–2014, he was an assistant professor at the University of Electro-Communications. He is currently an associate professor at Kochi University of Technology. His work is in the area of implementation of programming languages with specific interest of memory management. He received IPSJ Yamashita SIG Research Award in 2012.



Tatsuya Abe was born in 1979. He received his B.Sc. and Ph.D. degrees from Kyoto University and the University of Tokyo in 2002 and 2007, respectively. He worked for National Institute of Advanced Industrial Science and Technology, Kyoto University, and RIKEN. He is currently a senior research scientist at STAIR Lab, Chiba Institute of Technology. His research interests include programming languages, program verification, concurrency, and distributed computation. He is a member of IPSJ, ACM, and JSSST.