Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Visualization of Counterexamples of Memory Model-aware Model Checking Using SPIN

Kosuke Matsumoto[1,a)] Tomoharu Ugawa[1,b)]

Received: December 14, 2018, Accepted: March 17, 2019

Abstract: In modern computer processors, the order of memory access instructions in a program (the "program order") is not necessarily the same as the order in which the instructions are reflected in memory (the "memory order"). This means that the memory order has to be taken into account when verifying concurrent programs. We previously developed a library, MMLib, that facilitates model checking of programs by enabling the memory order to be taken into account when using the SPIN model checker. The input to SPIN is a model of the program written in Promela, an imperative style modeling language. When SPIN detects an error, it generates a trace of the execution path that led to the error. To determine what caused the error, the user has to understand this counterexample. However, counterexamples of models written using MMLib are difficult to understand because include internal MMLib execution steps, and the user has to interpret the internal data structures of MMLib to understand the memory order. We have now In this paper, we developed software for visualizing counterexamples of models written using MMLib. It visualizes execution of the model step by step along the path of the counterexample. It skips visualizing the internal MMLib steps and visualizes only the execution of each step corresponding to statement in the user model. It also highlights memory access instructions that have been executed but have not yet been reflected in memory. This makes users aware of when the memory order differs from the program order.

Keywords: SPIN, memory model, model checking, debug, visualization

1. Introduction

When model checking a concurrent program, not only does the order of memory access instructions written in the program, the *program order*, have to be considered but also the order in which they are reflected in memory, the *memory order*, has to be considered. This is because the two orders may be different in modern computer processors. Because memory access instructions executed in a thread are observed by other threads as if they were executed in memory order, a program that runs correctly if memory access instructions are reflected in memory in program order may not run correctly if the two orders differ. Possible memory orders are defined in the *memory model* of each processor.

We previously developed a library, MMLib, that enables the SPIN model checker to check programs in accordance with the memory model [6], [7]. MMLib is a set of model components written in Promela, a modeling language for SPIN. The user gives a model of the program together with MMLib to SPIN, and SPIN checks the model in accordance with the memory model.

MMLib is designed to be used for finding and fixing bugs caused by differences between the memory order and program order. It is intended for use after other tools are used to ensure that the program is error-free in regards to the program order. Therefore, we assume that the programs being checked using MMLib are error-free in terms of program order execution.

SPIN's modeling language, Promela, is an imperative language. It checks if an error might occur if thread executions are interleaved [*1]. It executes its statements of threads one by one. If an error could occur, the program is said to have a bug. When SPIN finds a bug, it generates a counterexample, which is a trace of the execution path leading to the error.

When debugging a program using model checking, a programmer should be able to understand the cause of the bug from the counterexample. However, there are two difficulties in understanding counterexamples when using MMLib. First, the counterexamples include the MMLib internal execution steps. Because MMLib is written in Promela, the same language used to write the program model, SPIN handles the execution steps of the program model and the MMLib internal execution steps in the same way when it generates a counterexample. This makes it difficult for a user to find the line in the model corresponding to an execution step in the counterexample and vice versa. Second, the user needs to interpret the values of the MMLib internal variables to understand the order in which memory access instructions are reflected in memory. The first difficulty is not specific to MMLib but is common among similar model checking libraries that are implemented using macros. The second one is specific to memory model-aware debugging. We focused on the second one and developed a way to visualize the memory orders.

From our experience in using memory model-aware model checking for debugging, we identified the following information for each execution step as useful for understanding bugs [5], [15].

[1] Kochi University of Technology, Kami, Kochi 782–0003, Japan
[a)] matsumoto@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp

[*1] Threads are called "processes" in Promela.

(1) Values of global and local variables in memory and the values observed by each thread.
(2) Memory access instructions that had been executed but were not yet reflected in memory.

We thus developed a counterexample visualizer that provides this information. The software reproduces the execution of the model along the path of the counterexample and displays this information on the model. More specifically, it shows step-by-step execution of the model at the statement or line level, like typical debuggers for imperative languages. Moreover, it highlights the memory access instructions that have not yet been reflected in memory and those that were reordered with subsequent instructions.

We explain the problem with debugging using MMLib in Section 2 and describe the memory models supported by MMLib and the structure of MMLib in Section 3. We then describe the design the counterexample visualizer and explain its operation in Section 4. In Section 5, we demonstrate debugging and bug understanding using the counterexample visualizer and in Section 6, we compare our counterexample visualizer with an existing one. We describe related work in Section 7 and summarize the key points and mention future work in Section 8.

2. Problem with Debugging Using MMLib

2.1 Peterson's Mutual Exclusion Algorithm

Figure 1 shows a program written using Peterson's mutual exclusion algorithm [10]. This algorithm is based on the assumption that memory access instructions are reflected in memory in program order. For memory models not based on this assumption, it fails to maintain mutual exclusion.

The program in Fig. 1 is based on the assumption that functions `T0` and `T1` are executed in different threads. The comment `Critical Section` represents the critical section for mutual exclusion. Shared variables `want0` and `want1` are used by threads `T0` and `T1`, respectively, to declare that the thread wants to enter the critical section. Hereafter, we refer to both of these variables as `want`. A thread attempting to enter the critical section sets `want` to 1. Then it tests the `want` of the other thread. If it is 0, the thread enters the critical section. Because a thread reads the other's `want` *after* it sets `want` to 1, it is impossible for both threads to be in the critical section simultaneously. However, there could be a deadlock if both threads set `want` to 1 and wait for the other one to clear it. The variable `turn` is used to prevent this deadlock.

```
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 Program using Peterson's mutual exclusion algorithm.

2.2 Promela Model Using MMLib

Figure 2 shows a model of the program shown in Fig. 1; it was written in Promela using MMLib. In Promela, a thread is defined with the keyword `proctype`. In the model in Fig. 2, threads `T0` and `T1` model the functions `T0` and `T1` in Fig. 1. Line 38, containing the keyword `ltl`, in the model in Fig. 2 has a linear temporal logic (LTL) formula specifying that both threads are never in the critical section simultaneously. SPIN searches for an execution trace that violates this specification.

A user can use MMLib to model threads accessing variables that are accessed by multiple threads. We call these variables *shared variables*. MMLib manages shared variables and provides macros to access them. That is, in MMLib, shared memory and memory access instructions are modeled using shared variables and *shared variable access instructions*, respectively. MMLib provides three macros as shared variable access instructions: `READ(`x`)`, which reads a value from shared variable x, `WRITE(`x`,` v`)`, which writes value v to x, and `FENCE()`, which forces preceding instructions to be reflected in the shared variables before the subsequent instructions are reflected.

```
1  #define PROCSIZE 2
2  #define VARSIZE 3
3  #define BUFFSIZE 5
4  #include "pso.h"
5  #define want0 0
6  #define want1 (want0 + 1)
7  #define turn  (want1 + 1)
8
9  proctype T0() {
10   WRITE(want0, 1);
11   WRITE(turn, 1);
12   if
13   ::(READ(turn) == 0)  -> skip;
14   ::(READ(want1) == 0) -> skip;
15   fi;
16 CS: skip; /*Critical Section*/
17   WRITE(want0, 0);
18 }
19
20 proctype T1() {
21   WRITE(want1, 1);
22   WRITE(turn, 0);
23   if
24   ::(READ(turn) == 1)  -> skip;
25   ::(READ(want0) == 0) -> skip;
26   fi;
27 CS: skip; /*Critical Section*/
28   WRITE(want1, 0);
29 }
30
31 init {
32   atomic {
33     run T0();
34     run T1();
35   }
36 }
37
38 ltl prop {! <> (T0@CS && T1@CS)}
```

Fig. 2 Promela model with MMLib for program in Fig. 1.

```
1  19: proc 2 (T0:1) peter.pml:11 (state 16) [temp = 0]
2              shared_memory[0] = 0
3              shared_memory[1] = 0
4              shared_memory[2] = 0
5              ...
6              buffer[6] = 1
7              buffer[7] = 0
8              buffer[8] = 1
9              ...
10             lcounter[6] = 1
11             lcounter[7] = 0
12             lcounter[8] = 1
13             ...
14 21: proc 2 (T0:1) peter.pml:14 (state 21) [(((( ((lcounter[((_pid*3)+1)]==0)) ->
15                                             (shared_memory[1]) : (buffer[((_pid*3)+1)]) )==0))]
16             queue 1 (queue[0]):
17             ...
18             queue 10 (queue[9]):
19             queue 11 (queue[10]): [1]
20             queue 12 (queue[11]): [0]
```

Fig. 3 Relevant portion of counterexample for model in Fig. 2.

In MMLib, shared variables are identified by an integer. Hence, `WRITE` expects an integer that identifies a shared variable as its first argument. Lines 5–7 in the model in Fig. 2 define names of shared variables written as C language macros, which is valid in a Promela model. We can thus use the names of shared variables as the first argument of `WRITE`. For example, `WRITE(turn, 1)` on line 11 writes 1 to the shared variable `turn`. Similarly, `READ(turn)` on line 13 reads from `turn`.

2.3 Cause of Error

In order for thread `T0` in the model in Fig. 2 to enter the critical section, at least one of the conditions of the `if` statement on line 12 must be satisfied. In Promela, a conditional branch can be described using `if` and `fi`. The conditions of an `if`-statement follow to the keyword "`::`"; it represents a nondeterministic choice. An `if`-statement chooses a branch for which the condition is true nondeterministically, if any, and executes it. If there is no such branch, the thread is blocked until one of the conditions is satisfied.

In the model in Fig. 2, thread `T0` reads `want1` on line 14; `want1` is the shared variable with which thread `T1` declares that it wants to enter the critical section. If the value written by `WRITE(want1, 1)` of thread `T1` on line 21 were visible to `READ(want1)` of thread `T0` on line 14 through the shared variable immediately after `T1` wrote the value, `T0` would be blocked by the `if`-statement, and mutual exclusion would be maintained. However, if there is a delay in reflecting the `WRITE` to the shared variable `want1`, `T0` enters the critical section because it observes that `want1` is still 0.

2.4 Counterexample

Figure 3 shows a relevant part of the counterexample generated by SPIN for the model in Fig. 2. In our previous research [5], counterexamples like this were interpreted to understand bugs. However, this required knowledge of the internal data structure of MMLib, and even with such knowledge, it was a difficult task. In the following, we explain how the information needed for understanding bugs related to memory order can be obtained.

2.4.1 Values of Shared Variables Being Accessed

The value of the shared variable being accessed by `READ` or `WRITE` depends on the values of multiple internal variables of MMLib. For example, the value read by `READ(want1)` on line 14 in the model in Fig. 2, which corresponds to line 14 of the counterexample in Fig. 3, depends on the values of `buffer`, `lcounter`, and `shared_memory`, the values of which are shown on lines 2–13 in the counterexample. In this example, `READ` reads the value of `shared_memory[1]`, which is 0, because `lcounter[7]` is 0, where the index of `shared_memory`, 1, is the integer identifying the shared variable `want1`, and the index of `lcounter`, 7, is computed using the integer.

2.4.2 `WRITE`s That Have Not Been Reflected

`WRITE`s that have not been reflected in shared variables are stored in MMLib's internal queues. The `queue`s shown on lines 16–20 in the counterexample in Fig. 3 store the `WRITE`s that have been executed but are not yet reflected in the shared variables. The element of `queue 11`, "`[1]`", represents `WRITE(want1, 1)` executed by `T1`. Because the `WRITE` has not yet been reflected in the shared variable, 0 is read from `want1` on line 14 in the model in Fig. 2. This sort of interpretation is needed to understand the counterexample.

3. MMLib

3.1 Memory Models

For models with MMLib, SPIN checks the executions in which shared variable access instructions are reflected in the shared variables in an order that differs from the program order. It performs model checking in accordance with the memory model. The current version of MMLib supports total store ordering (TSO) and partial store ordering (PSO) memory models. In TSO, only the pairs of a `WRITE` and a following `READ` can be reordered. In PSO, two `WRITE`s can be reordered in addition to TSO. In either memory model, reflection in the shared variables can be delayed only for the `WRITE`s.

Hereafter, we assume PSO memory model.

3.2 Structure of MMLib

In a model using MMLib, queues for each thread and the mem-

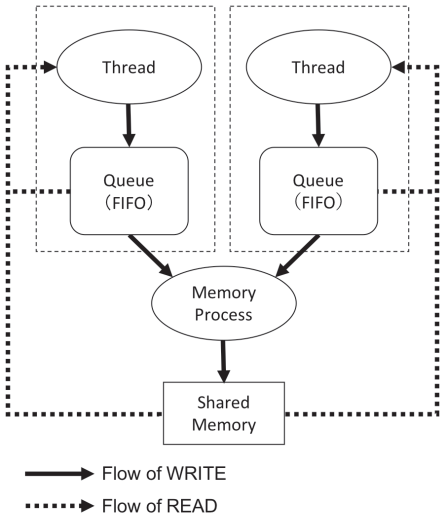

Fig. 4 Structure of model using MMLib.

ory process are introduced, as shown in **Fig. 4**. The "threads" represent those of the Promela model described by the user. The arrows represent the flows of data caused by a `WRITE` or a `READ`. Array `shared_memory` in Fig. 4 models shared variables. The indexes of the array are the integers used to identify the shared variables.

When a thread executes `WRITE`, a pair of an integer that identifies the shared variable to be written to and the value to be written is inserted into the queue as a write request. The (FIFO) queues are implemented using several channels in Promela. Their structures depend on the memory model. A write request inserted in a queue is dequeued by the memory process and reflected in the shared variable. The memory process nondeterministically chooses a queue from which it dequeues a write request. This enables an execution with a memory order that differs from the program order to be checked.

In contrast, `READ` completes immediately when it is executed. That is, `READ`s are reflected in the shared variables immediately. The value read by a `READ` may differ from one thread to another. If a write request is stored in the queue of the thread that executes the `READ`, the `READ` returns the value found in the write request. This mechanism uses the variable `lcounter` to track the number of write requests stored in each queue and the variable `buffer` to store the value in the latest write request.

When a thread executes a `FENCE`, all the write requests stored in the queue of the thread are immediately reflected in the shared variable. This causes the shared memory access instructions executed before and after the `FENCE` to be reflected in the shared variables in this order.

Shared memory access instructions are implemented using `atomic` blocks of Promela so that they can be executed without being interleaved with other threads during their execution. Thus, the lines that contribute to a single memory access instruction are contiguous in the counterexample.

4. Counterexample Visualizer

The counterexample visualizer we developed enables users to understand bugs without interpreting the counterexample. There were two specific requirements for this visualizer.

(1) It should visually display the time when each `WRITE` was reflected in a shared variable.
(2) It should display the values of the variables being accessed by shared variable access instructions.

We implemented this visualizer as a plug-in for the Eclipse.

4.1 Functions of Counterexample Visualizer

Our counterexample visualizer visualizes the execution of the model step-by-step like GUI debuggers for imperative languages. It shows information for each execution step, one by one. The execution step displayed is the *current execution step*.

An images of the counterexample visualizer displaying the counterexample for the error found in the model shown in Fig. 2 is shown in **Fig. 5**. The visualizer has a model view, a navigation view, and two variable views: a shared variable view and a local variable view, which is not shown in the figure. The model view shows the program model created by the user. The navigation view has buttons for loading a model and a counterexample and navigation buttons for moving the current execution step forward and backward. The shared variable view and the local variable view display the values of shared and local variables, respectively. Various types of information, such as the lines being executed by each thread at the current execution step and the *unreflected `WRITE`s*, which are the `WRITE`s that have yet to be reflected in the shared variables are displayed.

This counterexample visualizer has two functions corresponding to requirements (1) and (2) above.

(1) A function for highlighting unreflected `WRITE`s.
(2) A function for displaying the values of the shared variables observed by each thread.

4.1.1 Highlighting Unreflected `WRITE`s

The counterexample visualizer highlights in yellow in the model view the unreflected `WRITE` instructions at the current execution step. In accordance with our assumption that the program being checked is error-free in terms of program order execution programs being checked using MMLib are error-free in terms of program order execution, the bug the user is trying to understand using this counterexample visualizer is related to memory order. To understand such bugs, users have to focus on these unreflected `WRITE`s because their memory order is still uncertain.

The counterexample visualizer also highlights in red in the model view the unreflected `WRITE`s at the execution step when they match one of the following conditions because they can lead to bugs.

A. An unreflected `WRITE` W_0 preceding a shared variable access instruction, i.e., `READ` or `WRITE`, M in the program order, and M is reflected in memory at the execution step.
B. An unreflected `WRITE` W_1 writing to x, which x is a shared variable that *reflected* `READ` in another thread read from and the `READ` is reflected in the shared variable at the current execution step, and some shared variable access instructions following W_1 in the program order have been reflected in the shared variable.

Condition A can lead to a bug because the order of `WRITE` W_0

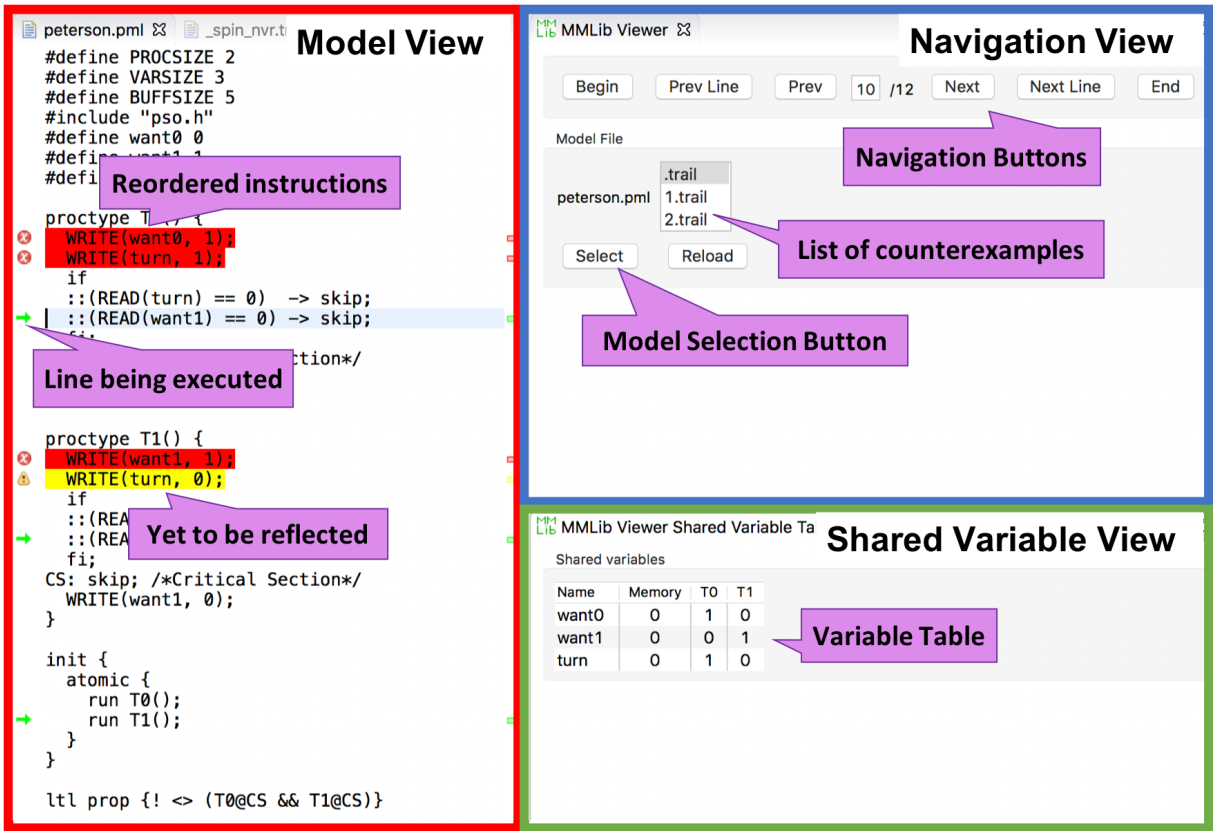

Fig. 5 Counterexample visualizer.

and M is changed, and this reordering occurs in the current execution step. It is important to note that executions of `READ`s are reflected in the shared variables immediately in all of the memory models supported by MMLib. In the model in Fig. 2 for example, when thread `T1` executes `READ(want0)` on line 25, `WRITE(want1, 1)` on line 21 is highlighted in red if it has not yet been reflected in the shared variable.

Condition B can lead to a bug because the reordering of W_1 with its following shared memory access instructions is observed by another thread. Note that W_1 was highlighted in a previous execution step due to condition A. In the model in Fig. 2 for example, when thread `T0` executes `READ(want1)` on line 14, `WRITE(want1, 1)` of thread `T1` is highlighted in red if it has not yet been reflected in the shared variable, and if the following `READ(want0)` has been executed. This is the cause of the error mentioned in Section 2.3.

4.1.2 Indicating the Values of Shared Variables Observed from the Viewpoint of Each Thread

As shown in Fig. 5, the shared variable view displays a table indicating the names of the shared variables and their values of the variables stored in them at the current execution step. The first column shows the names, and the second column shows their values at the current execution step. The following columns show the values observed from the viewpoint of each thread. The values observed from the viewpoint of a thread may differ from the values stored in the variables. The counterexample visualizer reproduces these observed values by using the same computation used by the `READ` macro.

The counterexample visualizer uses a *shared variable mapping file*, as described below in Section 4.2.2, to display the names of the shared variables in the first column of the table. If a shared variable mapping file is not given, the visualizer display the integers identifying the shared variables instead of their names.

The counterexample visualizer provides a local variable view in addition to the shared variable view. Because the memory order does not affect local variable accesses, its design is straightforward.

4.2 Other Features

4.2.1 Execution Steps

The counterexamples generated by SPIN include the MMLib internal execution steps. These steps are obstacles to debugging a program. The counterexample visualizer thus regards only events in the Promela model that the user described as execution steps. More specifically, an execution of a `WRITE` or `FENCE` macro is regarded as a single step though these macros comprise multiple atomically executed Promela statements. Reflecting an executed `WRITE` is also regarded as a single step though a dedicated process executes multiple atomically executed Promela statements to reflect it.

Promela has a dedicated thread called *never claim* for specifying a property that the model should *not* satisfy. The specification is written in an LTL formula and converted into a *never claim* thread. The thread is executed concurrently with other threads so that, if the model satisfies a property that should not be satisfied, the *never claim* thread enters a cycle that has an accept label. When the *never claim* thread enters such a cycle, SPIN regards it

as an error and creates a counterexample [*2]. Although the *never claim* thread is not a model of the program, its behavior is important for debugging the program. The counterexample visualizer thus deals with each step of the *never claim* thread as a single execution step.

4.2.2 Shared Variable Names

Although MMLib identifies shared variables by using integers, which we call *variable numbers*, users usually define symbolic identifiers for the variable numbers and use them to refer to shared variables in Promela models using MMLib. However, referring to the definitions while reading counterexamples is a tedious task. The counterexample visualizer accepts a shared variable mapping file, in which the user describes this mapping. The visualizer uses the names described in the mapping file for the names of the shared variables displayed in the shared variable view.

5. Case Studies

In this section, we describe debugging and bug understanding using the counterexample visualizer. We first describe the debugging of Peterson's mutual exclusion algorithm (Fig. 1) [*3]. We then describe how we can get an understanding of the bug in Staccato [8], a concurrent copying garbage collection (GC).

Table 1 shows the sizes of the models discussed here in terms of number of lines, the number of threads, and number of shared variables.

5.1 Strategy for Understanding Bugs

The strategy used for understanding bugs is as follows. First, a short counterexample is used. In general, if there is an error, there are multiple execution traces leading to the error. SPIN has an operation mode in which it searches for the execution trace with the smallest number of execution steps. Because a shorter counterexample is easier to understand, we use the shortest counterexample to understand the bug. The shortest counterexample is the shortest one including the MMLib internal execution steps as well. Nevertheless, it is sufficient as an approximation of the shortest counterexample for the Promela model described by the user.

Next, attention is paid to the `WRITE`s highlighted in red, which are the `WRITE`s that can lead to a bug. We assume that the program being checked runs correctly if its memory access instructions are reflected in the program order. This means that a bug is caused by reordering of the memory access instructions; that is, the memory access instructions are executed in an order different from the program order. The red highlighting visualizes reordering at the execution steps when shared variable access instructions are reflected in the shared variable in an order different from the program order.

Table 1 Model sizes.

model	no. of lines	no. of threads	no. of shared variables
Peterson	38	2	3
Staccato	174	2	6

[*2] SPIN translates a Promela model into a Büchi automaton and uses it to search for errors. In the semantics of a Büchi automaton, the automaton accepts an input if the control enters a cycle having an accept state. When the *never claim* is accepted, SPIN reports an error.

[*3] Strictly speaking, this is not a bug in the algorithm but misuse of the algorithm because Peterson's algorithm does not support memory models in which the program and memory orders are differ. However, we call it debugging here to modify the algorithm so that it can work on such memory models.

5.2 Peterson's Mutual Exclusion Algorithm

As we mentioned in Section 2, Peterson's mutual exclusion algorithm (Fig. 1) allows two threads to be in the critical section simultaneously if memory access instructions are executed in an order different from the program order. We demonstrate debugging of the model of Peterson's mutual exclusion algorithm (Fig. 2) using the counterexample visualizer.

(1) We loaded the shortest counterexample into the counterexample visualizer and advanced the execution to the execution step at which the first thread entered the critical section. We found that thread `T1` entered the critical section in this execution step because condition `READ(want0) == 0` on line 25 in the model in Fig. 2 was satisfied. This was the first step when `WRITE`s were highlighted in red (**Fig. 6**). In particular, `WRITE(want1, 1)` of thread `T1` on line 21 was highlighted in red at this step. This indicated that the declaration by thread `T1` to enter the critical section was still not visible to the other threads. This can lead to an error.

(2) In the following execution steps, thread `T0` progressed and executed a guard for the critical section `READ(want1) == 0` on line 14 in the model in Fig. 2 (**Fig. 7**). At this execution step, `WRITE(want1, 1)` on line 21, which thread `T1`

```
proctype T0() {
  WRITE(want0, 1);
  WRITE(turn, 1);
  if
  ::(READ(turn) == 0)  -> skip;
  ::(READ(want1) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want0, 0);
}

proctype T1() {
  WRITE(want1, 1);
  WRITE(turn, 0);
  if
  ::(READ(turn) == 1)  -> skip;
  ::(READ(want0) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want1, 0);
}
```

Fig. 6 Execution step in which thread `T1` enters critical section.

```
proctype T0() {
  WRITE(want0, 1);
  WRITE(turn, 1);
  if
  ::(READ(turn) == 0)  -> skip;
  ::(READ(want1) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want0, 0);
}

proctype T1() {
  WRITE(want1, 1);
  WRITE(turn, 0);
  if
  ::(READ(turn) == 1)  -> skip;
  ::(READ(want0) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want1, 0);
}
```

Fig. 7 Execution step in which thread `T0` enters critical section.

```
proctype T0() {
  WRITE(want0, 1);
  WRITE(turn, 1);
  FENCE();
  if
  ::(READ(turn) == 0)  -> skip;
  ::(READ(want1) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want0, 0);
}

proctype T1() {
  WRITE(want1, 1);
  WRITE(turn, 0);
  FENCE();
  if
  ::(READ(turn) == 1)  -> skip;
  ::(READ(want0) == 0) -> skip;
  fi;
CS: skip; /*Critical Section*/
  WRITE(want1, 0);
}
```

Fig. 8 Execution step in which first red highlight is displayed in new counterexample.

had executed, was highlighted in red, as well as the WRITEs executed by thread T0. This highlighting indicated that the declaration of thread T1 to enter the critical section was not observed by thread T0. In fact, the shared variable view indicated that the value of the shared variable want1 was still 0.

(3) From the results obtained above, we determined that the error was caused by the reordering of WRITE(want1, 1) on line 21 and READ(want0) on line 25 in the model in Fig. 2. We thus inserted a FENCE instruction before the if-statement on line 23 to prevent them from being reordered. Similarly, we inserted a FENCE in thread T0 because these two threads are symmetric. After fixing this bug in this way, we rechecked the model and found another error, for which we created a counterexample.

(4) We loaded this new counterexample into the counterexample visualizer and advanced the execution. **Figure 8** shows the first execution step in which a WRITE was highlighted in red. The thread that made progress at this execution step was T1, indicating that this highlighting was due to WRITE(turn, 0) on line 22 in the model in Fig. 2 being reflected in the shared variable before the preceding WRITE(want1, 1) on line 21. At this execution step, WRITE(turn, 1) executed by thread T0 on line 11 was highlighted in yellow, meaning that this WRITE had not been reflected in the shared variable. Once this WRITE(turn, 1) was reflected in the shared variable, and turn changed to 1, T1 was allowed to enter the critical section. Meanwhile, WRITE(want1, 1) executed by thread T1 on line 21 had not been reflected in the shared variable. Thus, thread T0 was also allowed to enter the critical section. In fact, in the following execution steps, thread T0 entered the critical section, and then thread T1 entered the critical section because condition READ(turn) == 1 was satisfied.

(5) The cause of this error was that thread T1 wrote to turn before the WRITE writing to want1 was reflected in the shared variable. This meant that thread T0, which was allowed to enter the critical section, overwrote turn, and thread T1 was allowed to enter the critical section. We thus inserted a FENCE between WRITE(want1, 1) on line 21 and WRITE(turn, 0) on line 22 to prevent these WRITEs from being reordered. We also inserted a FENCE in thread T0 in the same way. After fixing this bug in this way, we rechecked the model again. No more errors were found.

```
    if
    :: (forward == FROM_OBJECT) -> WRITE(from_data, val);
    :: else -> assert(READ(to_header) == TO_OBJECT);
      WRITE(to_data, val);
    fi
  :: else -> WRITE(to_data, val);
  fi
ewrite:
}

proctype collector()
{
    CAS_NORET(from_header, FROM_OBJECT, FROM_OBJECT_COPYING);
    start_handshake();
    endcol1:
    wait_for_handshake();

    AFENCE();
    WRITE(to_header, TO_OBJECT);
    WRITE(to_data, READ(from_data));
    RFENCE();
```

Fig. 9 Cause of bug in Staccato.

5.3 Concurrent GC Staccato

Staccato is a GC algorithm that manages the heap by dividing it into two semispaces: a from-space and a to-space. The collector copies all the objects in the from-space that can be accessed by the application threads (mutators) to the to-space. Concurrently with this copying, the mutators are using the objects. Thus, an appropriate synchronization is needed to ensure that the mutators access the up-to-date copies.

Staccato uses fence instructions so that it works correctly even if the memory order differs from the program order. However, there is a bug in Staccato: a value written by the mutator may not be copied to the to-space [15].

In an experiment, we developed a model of Staccato and reproduced the bug. We then, as described below, used the counterexample visualizer to understand the cause of this bug.

We generated the shortest counterexample by using SPIN and loaded it into the counterexample visualizer. Through the entire execution, three WRITEs out of 17 WRITEs in the model were highlighted in red 13 times in total. Among them, the WRITE(from_data, val) (highlighted in red on line 2 in **Fig. 9**) was the cause of the bug. This WRITE was executed by the mutator to write to the shared variable from_data, which represented the from-space. Because this write was delayed, the collector could not read the up-to-date value. As a result, the collector read an old value from from_data and copied it to the to-space using WRITE(to_data, READ(from_data)) on line 20. This was shown in the counterexample visualizer: because the collector read an old value from the shared variable from_data, the WRITE on line 2, which was writing to the same shared variable and had not been reflected, was highlighted in red.

5.4 Discussion

In the experiment on Peterson's mutual exclusion, red highlighting was useful. In the first counterexample, the execution steps in which WRITEs were highlighted in red were only the two mentioned above. At both of them, the WRITEs contributing to the error were highlighted. In the counterexample of the model after

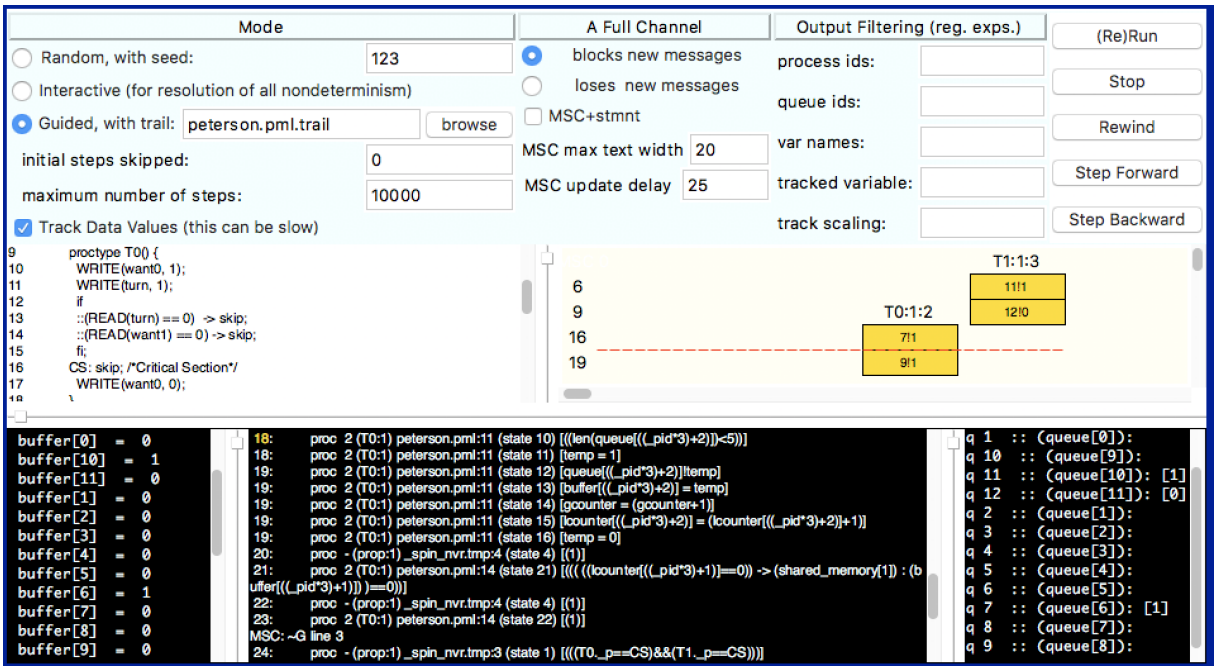

Fig. 10 iSPIN.

the first bug was fixed, although WRITEs were highlighted again in the steps following the steps mentioned above, the number of steps in which WRITEs were highlighted was small enough, and most of them were related to the error. The WRITEs contributing to the error were highlighted in the concurrent GC Staccato experiment as well. WRITEs were highlighted in red 13 times in total. This number is small enough for the user to examine each of them.

However, in the experiment on Peterson's mutual exclusion, the WRITE writing to turn, which did not contribute to the first error but the second error, was also highlighted in the execution steps mentioned above. This means that the causes of the bugs that we fixed in steps (3) and (5) above were indicated simultaneously. In the experiment on Staccato, the counterexample contained many unreflected WRITEs, and some of them were highlighted in red. In debugging of more complicated algorithms, the counterexample could contain more unreflected WRITEs.

The reason many WRITEs were reordered was that we used the counterexample that had the smallest number of execution steps as the shortest counterexample. Reflecting a WRITE in a shared variable takes multiple execution steps. And we had chosen a counterexample in which there was a delay in reflecting the WRITEs that did not contribute to the error in the shared variables. If we had chosen the semantically simplest counterexample rather than the shortest counterexample in which the MMLib internal execution steps were counted, it would have been easier to understand the bug. Addressing this problem remains for future work.

6. Comparison with Existing Counterexample Visualizer

iSPIN [11] is a standard counterexample visualizer for SPIN. Though iSPIN does not have any dedicated functions for MMLib, it is worth discussing whether iSPIN would be useful for debugging for users who have enough knowledge about the internal data structure of MMLib.

iSPIN shows step-by-step execution with line granularity of the counterexample. It shows all steps in the counterexample expect for those that execute the *never claim* thread. The iSPIN GUI (**Fig. 10**) displays a screen for navigating step-by-step executions in the upper part, views showing the model and sequence diagram in the middle part, and views showing the counterexample and the contents of variables and channels in the lower part. When debugging, the user reads the model shown without any highlighting or marking in the middle part while referring to the information shown in the other views. Here we compare some aspects of our counterexample visualizer with iSPIN.

6.1 Counterexample

As shown in Fig. 10, iSPIN shows the counterexample with almost no changes. For example, line 7 in the counterexample view is the same as line 1 in Fig. 3. Although the counterexample shown in Fig. 10 seems simpler than that in Fig. 3, this is because we generated the counterexample in Fig. 3 with more command-line options of SPIN. The counterexample in the view automatically scrolls so that the current execution step is shown in the view. Furthermore, the step number of the current execution step, which is at the left of the line, is colored yellow. Nevertheless, it does not indicate the lines being executed by each thread in the current execution step, unlike our counterexample visualizer does.

Our counterexample visualizer shows only a single execution step at a time in the display. In contrast, iSPIN shows the complete counterexample, which enables the user to check the execution steps around the current execution step. This is useful when visualizing counterexamples of models that do not use MMLib. However, for models that do use MMLib, the execution steps around the current execution step are of little use because the execution of a single shared variable access instruction is expanded

into multiple execution steps.

6.2 Values of Variables

iSPIN displays lists of variables and channels that occur in the counterexample at the left bottom and right bottom of the display. It updates the values of the variables and contents of the queues so that they match the current execution step. The user has to interpret this information to understand the values of the shared variables observed by each thread. However, even if the user has sufficient knowledge about the internal data structure of MMLib, it is difficult to understand the values for two reasons. First, it is difficult to find a single variable in a list containing the many variables used by MMLib. Second, it is even more difficult to understand the value of a shared variable because it depends on the values of multiple internal variables.

6.3 Filtering

In iSPIN, the user can specify regular expressions to filter the information to be displayed. The text boxes for these regular expressions are arranged on the left of the buttons located in the top left corner in Fig. 10. For example, if the user specifies "2" in the text box labeled "process ids," only the executions of the processes that have "2" in their IDs, which SPIN automatically assigns, are displayed.

This filtering function is useful for focusing on the behavior of a specific process or on the queue that stores the `WRITE` instructions to a specific shared variable. However, even with this function, it is difficult to hide the internal execution steps of MMLib in the counterexample in order to focus on the behavior of the Promela model of a program that uses MMLib.

6.4 Sequence Diagram

iSPIN shows a sequence diagram at the right in the middle part, as shown in Fig. 10. In this diagram, the operations to channels executed by threads are arranged in time sequence. Threads are arranged from left to right in the order of their IDs. Each yellow box in the diagram represents a single operation to a channel. For example, the topmost box indicates that thread `T1` inserted 1 in channel 11. The red dashed line indicates the current execution step.

MMLib uses channels to implement queues for storing unreflected `WRITE`s. Thus, a user who understands the implementation of MMLib can determine the times when `WRITE`s were inserted in and removed from the queues by interpreting the sequence diagram. If the sequence diagram indicated the names in the program model, such as the names of the shared variables, a user of our counterexample visualizer could benefit from the complementary use of such a sequence diagram. Future work includes adding a function for showing such a sequence diagram.

7. Related Work

Although there is a body of work on memory model-aware model checking, none of the proposed methods help the user to understand counterexamples, to the best of our knowledge. Abe et al. developed a memory model-aware model checker, McSPIN [1], that translates a program written in C into an internal representation. The internal representation is then translated into a Promela model that the user can use to check every possible execution path in accordance with the memory model specified by the user. SPIN produces an execution trace as a counterexample if it finds an error in the translated Promela model. However, the execution trace is in terms of the internal representation, so it does not correspond to the C program. Therefore, it is difficult for the user to use the execution trace for debugging. Travkin et al. developed a memory model-aware model checker, WEAK2SC [14], [16], that translates the internal representation of the LLVM compiler infrastructure generated from a C/C++ program into a Promela model that can emulate any possible execution in accordance with the memory model. The translated Promela model uses `goto` statements to control its execution. Thus, it is difficult to understand the counterexample and debug the C/C++ program. In contrast, our counterexample visualizer indicates the memory order of the Promela model specified by the user. The user can thus easily understand the counterexample.

Many counterexample visualizers have been developed that do not take the memory model into account. iSPIN [11], which we discussed above (Section 6), is the standard counterexample visualizer for SPIN. Pakonen et al. developed a model checker for hardware logics, MODCHK [9], that has a visualizer showing step-by-step execution on a model of the hardware logic. It also emphasizes the important part of LTL in understanding the counterexample. UPPAAL [3] is an integrated tool environment for modeling, simulation, and model checking for embedded systems. It visualizes the counterexample of model checking on a sequence diagram and a diagram of the automaton. Aljazzar et al. developed a tool, DiPro [2], that visualizes a counterexample of the PRISM and MRMC model checkers. It shows the execution on the automaton of the model. In contrast, our counterexample visualizer shows the model as is and visualizes the memory order on it because, in memory model-aware model checking under the assumption that the program being checked runs correctly in program order execution, the time when memory access instructions are reflected in memory is important.

Many work has focused on visualizing program execution. Terada et al. developed a debugging tool for students, ETV [13]. It visualizes the function calls in the execution log of a program on the program. Tanno et al. [12] also developed a debugger that visualizes information on the program. It visualizes the lines of the program that are executed frequently, which helps users debug interactive programs. Our counterexample visualizer also visualizes information on the user described model and visualizes the memory order.

Czyz et al. developed the JIVE debugging tool [4]. It generates call graphs, which represent the calling relationships of methods of classes, and sequence diagrams. This is useful for summarizing the behavior of programs, which facilitates the understanding of large-scale programs. However, in memory model-aware debugging, the focus of our work, it is more important to understand the fine-grain behavior of the program. Thus, our counterexample visualizer shows step-by-step execution of the counterexample on the Promela model.

8. Conclusion

We have developed a counterexample visualizer for the MMLib library that enables the SPIN model checker to check programs in accordance with the memory order. This counterexample visualizer shows step-by-step execution on the model along the path of the counterexample. This enables the user to obtain useful information for debugging, such as the order in which shared variable access instructions are reflected in shared memory, without having to interpret a counterexample of model checking using MMLib. As case studies, we used the counterexample visualizer to modify Peterson's mutual exclusion algorithm so that it can work under memory models that allow the memory order to differ from the program order and used the counterexample visualizer to understand the bug in concurrent GC Staccato. From these case studies, we found that it is useful to highlight reordered shared variable access instructions only at the execution steps in which these instructions are reflected in the shared variables.

Simple counterexamples must be generated to enable efficient debugging. In debugging using MMLib, we rely on SPIN's facility to generate the shortest counterexamples for generation of simple counterexamples. However, SPIN generates the shortest counterexamples on the model in which MMLib macros are expanded. Thus, they are not necessarily the simplest. In particular, in such counterexamples, shared variable access instructions tend to be reordered. This results in many shared variable access instructions being highlighted in our counterexample visualizer. The generation of semantically simple counterexamples remains for future work.

Acknowledgments We are deeply grateful to Haruto Tanno of Nippon Telegraph and Telephone Corporation for his advice in carrying out this research. This research was partially supported by JSPS KAKENHI Grant Number 16K00103.

References

[1] Abe, T. and Maeda, T.: A General Model Checking Framework for Various Memory Consistency Models, *Proc. 19th High-Level Parallel Programming Models and Supportive Environments*, pp.332–341 (2014).
[2] Aljazzar, H., Leitner-Fischer, F., Leue, S. and Simeonov, D.: DiPro – A Tool for Probabilistic Counterexample Generation, *Proc. 18th International SPIN Workshop on Model Checking Software*, pp.183–187 (2011).
[3] Behrmann, G., David, A., Guldstrand Larsen, K., Håkansson, J., Pettersson, P., Yi, W. and Hendriks, M.: Uppaal 4.0, *Proc. 3rd International Conference on the Quantitative Evaluation of Systems*, pp.125–126 (2006).
[4] Czyz, J.K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Proc. 2007 OOPSLA Workshop on Eclipse Technology eXchange*, pp.31–35 (2007).
[5] Iiboshi, H. and Ugawa, T.: Towards Model Checking Library for Persistent Data Structures, *Proc. 7th Non-Volatile Memory Systems and Applications Symposium*, pp.119–120 (2018).
[6] Matsumoto, K., Ugawa, T. and Abe, T.: Improvement of a Library for Model Checking under Weakly Ordered Memory Model with SPIN, *Journal of Information Processing*, Vol.26, pp.314–326 (2018).
[7] Matsumoto, K., Ugawa, T. and Abe, T.: A library of memory access instructions under relaxed memory models for SPIN, *Proc. 23rd Foundation of Software Engineering*, pp.63–72 (2016).
[8] McCloskey, B., Bacon, D.F., Cheng, P. and Grove, D.: Staccato: A Parallel and Concurrent Real-Time Compacting Garbage Collector for Multiprocessors, Research Report RC24504, IBM (2008).
[9] Pakonen, A., Buzhinsky, I. and Vyatkin, V.: Counterexample Visualization and Explanation for Function Block Diagrams, *Proc. 16th International Conference on Industrial Informatics*, pp.747–753 (2018).
[10] Peterson, G.L.: Myths about the mutual exclusion problem, *Information Processing Letters*, Vol.12, No.3, pp.115–116 (1981).
[11] spinroot.com: Getting Started: Using iSpin, spinroot.com (online), available from ⟨http://spinroot.com/spin/Man/3_SpinGUI.html⟩ (accessed 2018-12-14).
[12] Tanno, H. and Iwasaki, H.: Debugging Method without Suspending Program, *Proc. 35th JSSST Annual Conference* (2018).
[13] Terada, M.: ETV: A Program Trace Player for Students, *Proc. 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pp.118–122 (2005).
[14] Travkin, O. and Wehrheim, H.: Verification of Concurrent Programs on Weak Memory Models, *Proc. 13th International Colloquium on Theoretical Aspects of Computing*, pp.3–24 (2016).
[15] Ugawa, T., Abe, T. and Maeda, T.: Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models, *Proc. ACM on Programming Languages*, Vol.1, No.OOPSLA, pp.53:1–53:26 (2017).
[16] Wehrheim, H. and Travkin, O.: TSO to SC via Symbolic Execution, *Proc. 11th International Haifa Verification Conference*, 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 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.