SPIN で弱いメモリ順序のメモリモデルでのプログラムの実行をモデル検査するためのライブラリの改良

松元 稿如[1,a)] 鵜川 始陽[1,b)] 安部 達也[2,c)]

概要：現代の CPU は，プログラムのメモリアクセス命令の実行順序を入れ替える場合がある．しかし，モデル検査器 SPIN はそのような場合を検査しないため十分な検査を行うことができない．そこでこれまでに我々は，SPIN でメモリアクセス命令の順序が入れ替わる場合も検査できるライブラリを開発してきた．我々のライブラリを使えば，トータルストアオーダリング（TSO）とパーシャルストアオーダリング（PSO）に従って命令の順序が入れ替わる場合の検査が可能になる．このライブラリは，複数のスレッドで共有される変数（共有変数）のモデルと，共有変数を読み書きするためのマクロを提供する．しかし，このライブラリには三つの問題がある．第一に，SPIN では安全性や活性のような検査したい性質を LTL（Linear Temporal Logic）式で表現できるが，LTL 式内から我々のライブラリが提供する共有変数を参照できない．第二に，ある条件が成り立つまでスレッドの実行を止めるためによく利用されるガード文からも，共有変数の値を参照できない．最後に，共有変数の初期値は常に 0 になり，ユーザが指定することはできない．そこで本発表では，これらの問題を解決できるようにライブラリを改良した．改良したライブラリを用いていくつかのモデルを作成し，ライブラリの性能を調査した．

キーワード：SPIN，メモリモデル，LTL，モデル検査

An 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)]

Abstract: Modern multi-core CPUs may execute memory access instructions of programs out-of-order. However, the SPIN model checker does not check out-of-order executions, but it only checks in-order executions. We have developed a library for SPIN that enables to check 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. However, this library has three problems. Firstly, though SPIN accepts LTL (Linear Temporal Logic) 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 guard is not executable, could not refer to shared variables. Finally, the user could not specify initial values of shared variables, but they are initialised with zero. In this presentation, we improved the library to resolve these problems. We made models using our improved library and investigated the performance of the library.

Keywords: SPIN, memory model, LTL, model checking

[1] 高知工科大学
Kochi University of Technology, Kami, Kochi 782–0003, Japan
[2] 千葉工業大学人工知能・ソフトウェア技術研究センター
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

1. はじめに

モデル検査器 SPIN はプログラムの検査のために使用されている [6]．Promela という手続き型言語に近い逐次実行形式の言語によってプログラムの変数の操作や条件分岐などの動作をモデル化し，そのモデルから作成したオートマトンが全状態で検査したい性質を満たすかどうかを SPIN を使って網羅的に検査することでモデル検査を行う．

SPIN はモデル検査の際，プログラムのモデルに記述された順序の通りの実行を検査する．しかし，現代の CPU はメモリアクセス命令の完了を待たずに，その命令に依存しない後続命令を実行し，先に完了させることがある．マルチスレッドプログラムでこのような実行が起こると，別のスレッドからはプログラムと異なる順序でメモリアクセス命令を実行したように観測されることがあり，モデルに記述された順序通りの実行を検査するだけでは不十分である．どのような条件で先行命令の完了前に後続命令が完了することがあるかは，CPU アーキテクチャ毎に異なり，CPU のメモリコンシステンシモデル（以下，メモリモデル）によって定義されている．マルチスレッドプログラムのモデル検査では，メモリモデルに従ったあらゆる実行を検査する必要がある．しかし，プログラムに加えて，メモリモデルに従った実行まで含めたモデルを記述すると，モデルが煩雑になり，モデルに誤りが混入しやすい．

そこで我々はこれまでに，共有変数に対する読み書きが完了する順序に関して，SPIN でメモリモデルに従ったマルチスレッドプログラムの実行を検査できるようにするためのライブラリを開発してきた [15]．本ライブラリは，複数のスレッドからアクセスされる変数（共有変数）のモデルと，ユーザが記述したモデルから共有変数を読み書きするためのマクロを提供する．本ライブラリを使用することで，メモリモデルに従った動作を考慮せずにモデルを作成するのと同程度の手間で，メモリモデルに従ったあらゆる実行順序を検査できるモデルを作成できる．本ライブラリが提供するマクロは，共有変数を対象とする書き込みと読み込みの際に使用する．ユーザが共有変数に値を書き込む際には，`WRITE` マクロを使用する．`WRITE` は，第一引数の共有変数に第二引数の値を書き込むマクロであり，例えば共有変数 `x` に 1 を書き込むには `WRITE(x, 1)` と書く．ユーザが共有変数から値を読み込む際には，`READ` マクロを使用する．`READ` は，第一引数の共有変数の値を第二引数のローカル変数に格納するマクロであり，例えば，共有変数 `x` の値をローカル変数 `r` に読み込むには `READ(x, r)` と書く．

本ライブラリには三つの問題がある．一つ目の問題は，検査したい性質を線形時相論理（Linear Temporal Logic: LTL）式で記述する際に，共有変数を参照できないことである．プログラムの安全性や活性などの時間に関連した性質は LTL 式によって表現できる．SPIN でも，モデルが満たすべき性質をモデル中の変数や各プロセスの実行位置（プログラムカウンタ）を用いた LTL 式によって表現できる．例えば，共有変数 `want0` が 1 になれば，その時刻以降いつかはプロセス `t0` がラベル `CS` の位置に到達するという性質が常に成り立つことを表現する LTL は

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

と表すことができる．これは 2 章で示すピータソンの相互排除アルゴリズムのモデルで活性の性質を表している．しかし，本ライブラリで共有変数の値を使用するには，一旦 `READ` を使用してローカル変数に読み込まなければならないため，上記の LTL 式を記述できない．

二つ目の問題は，ガードからも共有変数を参照できないことである．Promela では，ガードと呼ばれる文を記述することで，ある条件が満たされるまでスレッドの実行を止めることができる．ガードを使うことで，モデルを簡潔に記述でき，モデル検査で探索する状態数も抑えることができる．例えば 2 章で示す **図 2** で使われている

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

というガードは，`turn == 0 || want1 == 0` の条件が満たされるまでブロックするという振舞いを簡潔に記述している．しかし，一つ目の問題と同じ原因，つまり，本ライブラリで共有変数の値を使用するには，一旦 `READ` を使用してローカル変数に読み込まなければならないため，共有変数を参照するガードも記述できない．そのため上記のガードは，下に示すような共有変数をポーリングするループに書き換える必要がある．

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

また，ガードが非決定的分岐の条件として用いられる場合，書換えが簡単ではないこともある．

三つ目の問題は，共有変数の初期値を設定できないことである．本ライブラリが共有変数に書き込む唯一の手段として提供している `WRITE` は，ユーザ定義プロセス内でしか使えない（詳細は 4 章参照）．そのため，ユーザ定義プロセスの起動前に共有変数を初期化する手段がない．その結果，共有変数の初期値は Promela の仕様上 0 となってしまう．

そこで本研究では，ライブラリを改良し，これらの問題

を解決した[*1]．改良したライブラリはメモリモデルに従った実行を検査できる既存のモデル検査器 [3], [12], [13], [14] に比べ，モデルの可読性が高い点，LTL で性質を記述できる点，軽量である点で優れている．

本論文では，まず予備知識として SPIN（2 章）とメモリモデル（3 章），改良前のライブラリ（4 章）について解説する．その後，本章で挙げた問題を解消できるようにライブラリを改良し（5 章），改良したライブラリを実装し，改良したライブラリの性能評価を行う（6 章）．さらに，メモリモデルや LTL 式を扱うモデル検査の関連研究について言及する（7 章）．最後に，本論文のまとめと今後の展望を述べる（8 章）．

2. SPIN

モデル検査器 SPIN は，モデル検査法に基づいてハードウェアやソフトウェア，プロトコルなどの自動検証を行うツールである [6]．

2.1 モデル検査法

モデル検査法とは，検査対象の状態が変化する動作に注目して記述されたモデルが取り得る状態を網羅的に検査する形式手法である [5]．例えばプログラムを検査する場合は，変数の操作や条件分岐などの動作を抽出したモデルを作成し，モデルの取り得る全状態，つまり計算機上で実行した時の変数の取り得る値やプログラムの実行位置などが期待通りであることを検査する．SPIN を使用してモデル検査するためのモデルは，専用の記述言語 Promela でプログラムと検査したい性質を記述し作成する．作成したモデルを入力として与えると，SPIN は与えられたモデルを Büchi オートマトンに変換し，そのオートマトンの全状態で検査したい性質が成り立つかどうかを網羅的に検査することでモデル検査を行う．

一般にモデル検査法には，有限の状態に収まる検査対象しか扱えないという欠点がある．例えば任意の自然数に対する実行について検査することはできない．他にも，モデルが大きくなるにつれて検査する状態数が指数的に大きくなるという欠点がある．そのため，モデルを作成する際には状態数を抑える工夫が重要になる．

2.2 Promela

Promela は C に似た構文を持ち，C のマクロを使用できる．マルチスレッドプログラムのモデル記述に向いており，スレッドに相当するプロセスを記述するためのコンストラクトを持つ．プロセス毎に記述された文は記述された順序通りに他のプロセスの文とインターリーブして実行される．Promela では，「;」もしくは「->」で区切られた式を文として扱う．モデル中のあらゆる文は真か偽かを判定され，偽の文は実行されない．偽の文が実行されないことで，プロセスの実行が止まることをブロックするという．

例として，**図 1** のピーターソンのアルゴリズムによって相互排除を行う C のプログラムを，Promela でモデル化する．まず，図 1 のプログラムについて説明する．図 1 のグローバル変数 `want0` と `want1` は，それぞれスレッド `t0` と `t1` がクリティカルセクションに入ろうとしていることを相手のスレッドに知らせるためのフラグである．グローバル変数 `turn` は，その時点で優先的にクリティカルセクションに入ることのできるスレッドを表している．`t0` がクリティカルセクションに入るためにはまず，`want0` を 1 にすることで，相手のスレッドに自身がクリティカルセクションに入ろうとしていることを知らせる．次に，`turn` を操作して相手のスレッドが優先的にクリティカルセクションに入れるようにする．その後，`want1` が 0 になるか，相手のスレッドの操作によって `turn` が 0 になるまで待ち状態になる．条件が整えば待ち状態を抜けてクリティカルセクションに入る．クリティカルセクションを抜けると，`want0` を 0 にすることで相手のスレッドに知らせる．

図 2 は 図 1 のモデルである．Promela ではスレッドに相当するプロセスを記述するために `proctype` というキーワードを使う．また，ユーザ定義のプロセスよりも先に実行される init プロセスが利用できる．ユーザ定義プロセスは 21 行目のように `run` 命令によって実行される[*2]．20 行目から 23 行目の `atomic { }` で囲まれた範囲は，ひとまとまりの命令として実行され，実行可能な限りインタリーブされない．繰り返しは `do` と `od` の間に記述するが，図 1 の `while` による待ち状態は，図 2 の 6 行目の文がブロックすることを利用してモデル化している．6 行目のような，続く文が実行されるための条件として記述される文をガードという．図 2 では使用していないが，条件分岐は以下のように `if` と `fi` の間に記述し，「`::`」の後のガードが真ならば「`->`」に続く文が実行される．

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

複数のガードが同時に真になった場合は，いずれかのガードに続く文が非決定的に実行される．ただし，「`::`」に続く文が全て偽ならブロックする．Promela では数値型の変数のほかにチャネル型の変数が用意されており，FIFO 形式のキューとして扱うことができる．他にも，LTL 式などで使用するためのラベルを記述でき，例えば，11 行目の `CS:`のように，ラベルを記述することもできる．

[*1] 本ライブラリと本論文で使用したモデルは，https://github.com/plasklab/mmlib で公開している．

[*2] `active proctype` というキーワードによって，自動で起動するプロセスも記述できるが，本研究では扱わない．

```
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 }
```

図 1 C で記述されたピーターソンのアルゴリズムによる相互排除プログラム

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 }
```

図 2 Promela で記述された 図 1 のプログラムのモデル

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

2.3 モデルの満たすべき性質の記述方法

SPIN でモデルが満たすべき性質を記述する方法には，assert 文で記述する方法と LTL 式を使う方法がある．

assert 文を使用する方法は，assert(ψ) という文をモデル中の任意の位置に挿入することである．ただし，ψ は以下で定義される論理式である．

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

ここで $\top$ は真であることを意味する命題定数である．P は SPIN で利用可能な原子命題を表す命題変数であり，例えば， 変数や即値の等号（==）や比較演算子（<）を用いて表現される命題である．詳しくは文献 [6] を参照されたい．$\neg\psi$ は ψ の否定を表し，$\psi_0 \vee \psi_1$ は ψ_0 と ψ_1 の論理和を表している．論理積や含意は略記として定義する．モデルに挿入された位置でその assert 文中の式が真ならばなにもせず，偽ならばエラーを出力し検査を打ち切る[*3]．

assert 文が挿入された位置においての真偽のみを扱うのに対し，LTL 式は，ある性質が与えられた時「その性質がいつか真になる」といった，（ある時点以降の）時間に関連した性質を表現するための論理式である．LTL 式 φ は以下で定義される論理式である．

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

$\mathrm{X}\,\varphi$ は「次の時点で φ が成り立つ」ことを意味している．$\varphi_0\,\mathrm{U}\,\varphi_1$ は「φ_1 が成り立つようになるまでは φ_0 が成り立ち続け，かつ，いつか φ_1 が成り立つ」ことを意味している．便利のため，以下の略記を使用する．

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

$\Diamond\varphi$ は「いつか φ が成り立つ」ことを意味し，$\Box\varphi$ は「ずっと φ が成り立ち続ける」ことを意味している．

LTL 式が意味するものの定義を以下で与える．SPIN が探索するモデルの実行経路の集合は一つの状態遷移系と考えられる．この状態遷移系をクリプキモデル $\mathfrak{M} = \langle M, R, val\rangle$ で書くことにする．ただし，M は状態の集合，R は M 上の関係，val は命題変数から M の部分集合への関数である．$\mathfrak{M}$ 上の**経路** π を，R で関係づけられている M の元の（有限とは限らない）列とし，π の i 番目（先頭は 0 番目とする）の状態を $\pi(i)$ ，π から i 番目の状態以降の列を π^i と書くことにする．$\mathfrak{M}$ 上の経路 π による LTL 式 φ の解釈 $\mathfrak{M},\pi \vDash \varphi$ を

*3 打ち切らないこともできるが，本論文では打ち切る場合のみを扱う．

$$
\begin{aligned}
\mathfrak{M}, \pi \vDash \top &\iff \text{真である} \\
\mathfrak{M}, \pi \vDash P &\iff \pi(0) \in val(P) \\
\mathfrak{M}, \pi \vDash \neg\varphi &\iff \mathfrak{M}, \pi \nvDash \varphi \\
\mathfrak{M}, \pi \vDash \varphi_0 \vee \varphi_1 &\iff \mathfrak{M}, \pi \vDash \varphi_0 \text{ または } \mathfrak{M}, \pi \vDash \varphi_1 \\
\mathfrak{M}, \pi \vDash \mathrm{X}\,\varphi &\iff \mathfrak{M}, \pi^1 \vDash \varphi \\
\mathfrak{M}, \pi \vDash \varphi_0 \,\mathrm{U}\, \varphi_1 &\iff \text{ある } j \in \mathbb{N} \text{ が存在して } \mathfrak{M}, \pi^j \vDash \varphi_1, \\
&\qquad \text{かつ，任意の } i < j \text{ に対して } \mathfrak{M}, \pi^i \vDash \varphi_0
\end{aligned}
$$

と定義する．この定義より，

$$
\begin{aligned}
\mathfrak{M}, \pi \vDash \Diamond\varphi &\iff \text{ある } i \in \mathbb{N} \text{ が存在して } \mathfrak{M}, \pi^i \vDash \varphi \\
\mathfrak{M}, \pi \vDash \Box\varphi &\iff \text{任意の } i \in \mathbb{N} \text{ について } \mathfrak{M}, \pi^i \vDash \varphi
\end{aligned}
$$

である．$\mathfrak{M}$ 上の $m_0 \in M$ を始点とする任意の経路 π による LTL 式 φ の解釈 $\mathfrak{M}, \pi \vDash \varphi$ が満たされるとき，$\mathfrak{M}, m_0 \vDash \varphi$ と書く．

Promela では，含意は `->`，$\Diamond$ は `<>`，$\Box$ は `[]` と書かれる．図 2 において，`want0 == 1` が成り立てば，いつかは `t0@CS` の位置に実行が移ることが常に成り立つという活性の性質は，

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

と記述する．

LTL 式で表された性質を SPIN で検査する場合，性質の否定を表す LTL を never プロセスという特殊なプロセスに変換する[*4]．never プロセスは他のプロセスとは独立に動作し，never プロセスが性質の否定を表す LTL を満たすと，never プロセスが表す Büchi オートマトンが受理状態を含むループに入り，エラーが出力され検査が打ち切られる．

3. メモリモデル

現代の CPU は最適化の際に，先行命令に依存しない後続命令を実行し，先に完了させることがある．その結果，マルチスレッドプログラムにおいて，それぞれのスレッドは，自身以外のスレッドが実行したメモリアクセス命令の列が，プログラムの順序とは異なる順序で実行されたかのように観測する場合がある．どのような条件で先行するメモリアクセス命令の完了前に後続のメモリアクセス命令を完了させることがあるかは，CPU アーキテクチャ毎に異なり，CPU のメモリモデルによって定義されている．

本研究では，メモリアクセス命令として書き込み命令（ストア）と読み込み命令（ロード）を考え，あるスレッドがメモリアクセス命令 A の後にメモリアクセス命令 B を実行したという順序を $A \rightarrow B$ と表すことにする．すると，二つのメモリアクセス命令の実行順序として，ストア → ストア，ストア → ロード，ロード → ストア，ロード → ロードの四つを考えることができる．本研究ではメモリモデルを，異なるメモリアドレスに対する上記の四つの実行順序について，それぞれ CPU が後続命令を先行命令の完了を待たずに実行し完了させることがあるかを定義したものとする．以降，あるスレッドが実行したメモリアクセス命令の列が，他の全てのスレッドからは異なる順序で実行されたかのように観測される場合があることを，**実行順序が緩和されている**という．

```
初期値 x = 0; y = 0;
スレッド A          スレッド B
Store(x, 1);       Store(y, 1);
Load(y, eax);      Load(x, ebx);
```

図 3 SC と TSO で実行結果が異なる場合のあるプログラム例

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

メモリアクセス命令の実行順序が緩和されているメモリモデルを採用しているアーキテクチャは，プログラムの動作がユーザの想定とは異なる場合がある．そのようなアーキテクチャでは，メモリアクセス命令の実行順序を強制する命令としてフェンス命令を提供している．フェンス命令を挿入することで，フェンス命令に先行するメモリアクセス命令は，必ずフェンス命令の後続のメモリアクセス命令より前に完了する．

本研究では，いかなる実行順序も緩和されないシーケンシャルコンシステンシ（SC）[8]，ストア → ロードの実行順序が緩和されているトータルストアオーダリング（TSO）[4]，ストア → ロードとストア → ストアの実行順序が緩和されているパーシャルストアオーダリング（PSO）[4] の三つのメモリモデルを扱う．**図 3** に，SC と TSO で実行結果が異なることがあるプログラムの例を示す．`Store(x, 1)` はメモリアドレス `x` に 1 を書き込むことを，`Load(y, eax)` はメモリアドレス `y` の値を `eax` レジスタに読み込むことを示す．例えば，スレッド A の 2 行目で `y` から 0 を読み込んだ場合を考えると，SC ではスレッド B の 2 行目で `x` から 1 を必ず読み込むが，TSO ではストア → ロードの実行順序が緩和されているため 0 を読み込むことがある．

4. 既存のライブラリ

本章では，改良前の本ライブラリについて解説する．

本ライブラリは，共有変数のモデルと，ユーザが作成したモデルからそれらに対してメモリモデルに従った読み書きを行うためのマクロを提供している．各スレッドは，共有変数を介してのみメモリモデルの影響を受け，CPU がメモリアクセス命令の列の順序を入れ替えて完了させた結果を観測する．よって，メモリモデルに従ったモデル化を行うのは，共有変数と，それらに対する読み書きのみで十分である．メモリモデルの影響を受けないローカル変数に対する読み書きは通常の Promela と同様に行う．以降，ユー

[*4] LTL 式を Promela モデル中に記述する方法もある．

ザが作成したモデルを**ユーザモデル**といい，ユーザモデル中の init 以外のユーザ定義プロセスを**ユーザプロセス**という．

本ライブラリが扱う SC，TSO，PSO の三つのメモリモデルで緩和されるメモリアクセス命令の実行順序は，ストア → ロードとストア → ストアのみである．どちらの実行順序も，後続のメモリアクセス命令が先行するストアより前に実行が完了する場合があることを表している．そのため，これらのメモリモデルに従った動作をするメモリアクセス命令をモデル化するためには，ストアの結果が後続のメモリアクセス命令よりも後にメモリに反映される場合があるようにすればよい．本ライブラリでは，ユーザプロセスが実行したストアの内容を格納し溜めておくための**ストアバッファ**と，ストアバッファの中身をメモリに反映する特別なプロセスである**メモリプロセス**を導入するアイデアによって，TSO と PSO に従った動作を実現している． なお，プログラムの実行順序とは異なる順序で実行されたかのように観測されるのは，自身以外のスレッドが実行したメモリアクセス命令だけである．つまり，自身が実行したストアは即座にロードできる．これを実現するために，各ユーザプロセスに自身が実行したストアの内容を記録しておく仕組みとして**共有メモリのコピー**を導入している．この仕組みを使って，自身が実行したストアで書き込まれた変数からの読み込みは，そのストアが共有メモリに反映されるまでは共有メモリを介さず，共有メモリのコピーからロードするようになっている．

この設計では，ユーザプロセスで atomic ブロックを実行中はストアバッファが一杯になってユーザプロセスがブロックする場合を除いて，メモリプロセスの実行が進まない．そのため共有メモリへの反映が行われないことになる．しかし，atomic ブロックを実行しているプロセスは自身のストアを共有メモリを介さずにロードできることと，それ以外のユーザプロセスは実行できないことから，atomic ブロック実行中にストアバッファの中身がメモリに反映されたとしても atomic ブロック実行中直後に反映されるのと同じ結果になる．

本ライブラリが提供するマクロを使用するには，ユーザモデルが持つプロセスの数や共有変数の数などをパラメタとして与える必要がある．具体的には，ユーザモデルが持つプロセスの数（PROCSIZE），共有変数の数（VARSIZE），ストアバッファのサイズ（BUFFSIZE）の三つを，ユーザモデルで本ライブラリをインクルードする前にマクロとして定義する．ただし，BUFFSIZE は同時に緩和できるストアの数の制限になっているため，BUFFSIZE が十分な大きさでない場合は，検査漏れが生じる危険がある．本ライブラリでは，これらのパラメタの設定はユーザの責任としている．

本ライブラリが提供しているマクロは以下の通りである．本ライブラリでは，共有変数を 0 から VARSIZE − 1 までの**メモリアドレス**でモデル化しており，これらのマクロの引数では共有変数の代わりにメモリアドレスを指定する．

WRITE(a, v)
: メモリモデルに従った動作をするストアを提供する．a はストア対象のメモリアドレス，v は書き込む値である．

READ(a, x)
: メモリモデルに従った動作をするロードを提供する．a はロード対象のメモリアドレス，x は読み込んだ値を格納したいローカル変数である．

FENCE()
: フェンス命令を提供する．

これら三つ以外に，実際の CPU では CAS（Compare and Swap）命令等のアトミックなメモリアクセス命令も提供されているが，本ライブラリではそれらに対応するマクロを提供しない．それでも，本ライブラリが提供する命令を組み合わせることで，いくつかのアトミック命令はユーザが作成することができる．例えば SPARC TSO の CAS 命令は Promela の atomic ブロックと READ，WRITE，FENCE 命令を組み合わせることで作ることができる．

以下では，本ライブラリが提供するメモリアクセス命令の実装を述べる．

どのメモリモデルでも共有変数の集合を配列として表している．

SC では，WRITE と READ は，それぞれ通常の Promela と同様に共有変数に対して書き込みと読み込みを行い，FENCE は何も動作をしない．

TSO では，ストア → ロードの実行順序が緩和されているため，WRITE が後続の READ よりも後にメモリに反映されることがある．そのために，メモリプロセスと FIFO 形式のストアバッファ，共有変数のコピー，各共有変数に対するストアが幾つストアバッファに格納されているかを数えるためのカウンタを導入する．ストアバッファと共有変数のコピー，カウンタはユーザプロセス毎に用意しており，それぞれプロセスのプロセス ID をインデックスとする配列になっている．メモリプロセスは，以下の手順をアトミックに実行し，各ユーザプロセスの書き込みを共有変数に反映する．

(1) いずれかのユーザプロセスが持つストアバッファを選択し，中身を先頭から一つだけ非決定的に取り出す．
(2) (1) で取り出したストアの結果を共有変数に反映する．
(3) (1) で選択したストアバッファを持つユーザプロセスのカウンタの，ストアを反映した共有変数に対応する値を減らす．

WRITE は図 4 に示すコードのように，実行したユーザプロセスのプロセス ID に応じて，ストアバッファ（queue）に引数のメモリアドレスと書き込みたい値の組を，引数

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

図 4 TSO の WRITE マクロの実装
Fig. 4 Implementation of WRITE macro for TSO.

```
1  #define(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 }
```

図 5 TSO の READ マクロの実装
Fig. 5 Implementation of READ macro for TSO.

```
1  #define(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  }
```

図 6 PSO の WRITE マクロの実装
Fig. 6 Implementation of WRITE macro for PSO.

```
1  #define(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  }
```

図 7 PSO の READ マクロの実装
Fig. 7 Implementation of READ macro for PSO.

のメモリアドレスに対応する共有変数のコピー（buffer）へ書き込みたい値を書き込み，カウンタ（counter）の値を一つ増やす，という手順をアトミックに実行する．ここで _pid は WRITE を実行したプロセスの ID，s はメモリアドレス，v は書き込む値である．READ は図 **5** に示すコードのように，実行したユーザプロセスが持つストアバッファの中身に応じて，引数のメモリアドレスに対応する共有変数（shared_memory）もしくは共有変数のコピーの値を引数のローカル変数に格納する，という手順をアトミックに実行する．メモリプロセスは，ユーザプロセスとインタリーブしながらストアバッファの中身を共有変数に反映するため，インタリーブの仕方によって先行するストアが後続のロードが実行された後で共有変数に反映され，ストア → ロードの実行順序を緩和できる．FENCE は，各プロセスが持つストアバッファの中身を直ちに全て共有変数に反映する．

PSO では，ストア → ストアの実行順序が緩和されているため，WRITE が後続のメモリアクセス命令よりも後にメモリに反映されることがある．そのために，ユーザプロセス毎かつ共有変数毎にストアバッファが用意されている．WRITE は図 **6** に示すように実行したユーザプロセスのプロセス ID に対応するストアバッファのうち，書き込み対象の共有変数に対応するストアバッファに書き込みたい値のみを格納する．ここで，gcounter はストアバッファに格納されているストアの総数を管理するカウンタである．READ は，TSO ではカウンタの値を確認しながら共有変数の値を読み込むように実装しているが，PSO ではチャネルの長さを取得できる len 関数を利用して実装している．そのため，PSO では図 **7** に示すようにカウンタは必要ない．FENCE は TSO と同様である．ストアバッファが共有変数毎に分かれているため，先行するストアと後続のストアの書き込み対象の共有変数が異なる場合は，インタリーブの仕方によっては先行するストアが後続のストアが共有変数に反映された後で共有変数に反映される場合があり，ストア → ストアの実行順序を緩和できる．ストア → ロードの実行順序については TSO と同様に緩和できる．

以下では，本ライブラリが提供するマクロを使用する方法を述べる．

図 **8** は，図 2 のモデルのうち，グローバル変数とプロセス t0 を本ライブラリを使用するように変更したモデルである．1 行目から 3 行目まででパラメタのマクロを定義している．プロセスは t0 と t1 の二つであるため，PROCSIZE は 2 と定義している．共有変数は want0 と want1，turn の三つであるため，VARSIZE は 3 と定義している．共有変数への書き込みの数はどのプロセスでも高々三回であるため，BUFFSIZE は 3 と定義している，続く 4 行目から 6 行目は，モデルの可読性を向上させるためのマクロである．共有変数は 0 から VARSIZE − 1 までの整数で表すため，4 行目から 6 行目のようにマクロを定義することで元のユーザモデルと同じ使い勝手で共有変数を扱える．7 行目でインクルードしている tso.h は，本ライブラリのうち，TSO に従った実行を検査するためのライブラリである．8 行目

```
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 }
```

図 8 改良前のライブラリを使用したピータソンの相互排除アルゴリズムのモデルの一部

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

以降は，共有変数を対象とする書き込みと読み込みを本ライブラリが提供するマクロを使用するように変更したモデルである．ただし， 図 2 の 8 行目から 10 行目は，本ライブラリでは共有変数を参照するガードを通常の Promela と同じように記述できないために図 8 の 14 行目から 24 行目のように繰り返しを含むモデルに書き換えた．

5. ライブラリの改良

本ライブラリには二つの制約がある．一つ目は，共有変数の値を読み出すには READ を使用して，一旦ローカル変数に格納しなければならないことである．そのため，例えば式の中で直接共有変数の値を参照できない．二つ目は，WRITE と READ は実行したプロセスのプロセス ID に依存した実装のため，ユーザプロセスでしか使用できないことである．例えば init プロセスや LTL 式では使用できない．本章では，これらの制約による問題を解決できる新しいマクロを提供するように，本ライブラリを拡張する．

ユーザが共有変数の値を参照する式を記述するには，一旦ローカル変数に格納し，そのローカル変数を組み込んだ式を評価することになる．しかし，各プロセスがインターリーブしながら実行されるため，式を評価する直前に共有変数の値を読み込んだとしても，式を評価した時に最新の値であるとは限らない．そのため，本ライブラリを使用すると，ユーザモデル中の共有変数を参照するガードをそのまま用いることはできず，図 8 の 14 行目から 24 行目のような書換えが必要になる．そこで本研究では，これまでの共有変数の値を一旦ローカル変数に格納する READ の代わりに，共有変数の値を直接返す新しい READ を提供する．これによりガードから共有変数を参照できるようになる．本研究では，以下のような新しい READ マクロを提供する．

READ(s)
: メモリモデルに従った動作をするロードを提供し，共有変数 s の値を返す．

共有変数は本ライブラリが管理しており，ユーザが共有変数に値を書き込むには WRITE を使用する必要がある．共有変数の初期値はユーザプロセスを実行する前に設定する必要があるが，WRITE はユーザプロセスでしか使用できない．そのため，共有変数の初期値を設定できず，共有変数の初期値は Promela の仕様により 0 となる．そこで本研究では，init プロセス中で共有変数を初期化できるように，以下のような初期値を設定するためのマクロを提供する．

INIT(s, v)
: 共有変数 s の初期値を v に設定する．

このマクロは，ストアバッファを介さずに共有変数の配列に直接書き込む．

LTL 式を使って検査を行う際にも，READ では直接共有変数の値を参照できないため，共有変数を参照した LTL 式は記述できない．さらに，READ は実行するユーザプロセスのプロセス ID に依存するため，新しい READ を使用しても共有変数を参照する LTL 式は記述できない．そこで本研究では，専用のマクロを提供することで，LTL 式から共有変数を参照できるようにする．ただし，メモリモデルによってはプロセス毎に観測できるメモリの値が異なる場合がある．そのため本研究では，以下のように，共有メモリに反映された共有変数の値を参照する GSVAR マクロに加え，ある特定のプロセスから観測できる共有変数の値を参照する SVAR マクロを提供する．

GSVAR(s)
: 共有変数 s の値を返す．

SVAR(p, s)
: ユーザプロセス p から観測できる共有変数 s の値を返す．

なお，共有変数への書き込みを一切行わないプロセスを用意すれば，そのプロセスが実行した SVAR の結果は常に GSVAR と同じになる．そのため，SVAR のみを使用して実際の共有変数の値を参照できる．しかし，そのために用意す

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

図 9 TSO の新しい READ マクロの実装
Fig. 9 Implementation of new READ macro for TSO.

るプロセスはユーザモデルと関係ない無駄なプロセスであるため，本研究ではそれぞれの共有変数の値を参照する仕組みを GSVAR マクロとして提供する．

以下では，新しく提供するマクロの実装と使用方法を述べる．

5.1 ガードから共有変数を参照するためのマクロ

本節では，ガードから共有変数を参照するための，共有変数の値を直接返す新しい READ の実装と使用方法を述べる．

SC では特別な工夫は必要なく，引数の共有変数の値を通常の Promela の方法で返すように実装する．TSO では，**図 9** のように，実行したプロセスのプロセス ID に対応するストアバッファに引数の共有変数を対象とするストアが格納されていなければ共有変数の値を，格納されていればプロセスのプロセス ID に対応する共有変数のコピーの値を返すように実装する．PSO では，実行したプロセスのプロセス ID に対応するストアバッファのうち，引数の共有変数に対応するストアバッファが空なら共有変数の値を，空でないなら実行したプロセスのプロセス ID に対応する共有変数のコピーの値を返すように実装する．

これまでの READ は，共有変数 x の値をローカル変数 a に読み出す際は，

READ(x, a);

のように使用したが，新しく提供する READ は，

a = READ(x);

のように使用する．新しい READ を使用することで，図 8 の 14 行目から 24 行目の繰り返しを含む待ち状態を，以下のように繰り返しを含まないよう 1 文でモデル化できる．

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

ただし，新しい READ は直接共有変数の値を返すために，WRITE の第二引数の中で使われる可能性がある．READ はストアバッファに要素が格納されているかどうかで動作が変わるため，WRITE の第二引数に WRITE の書き込み先である変数を読みだす READ が使われると問題が起こる．図 4 に示した TSO の WRITE の実装では，ストアバッファに格納する時と共有変数のコピーに格納する時で READ で読み込む箇所が異なるからである．そこで，本研究では TSO で**図 10** のように第二引数を一旦ローカル変数に格納するように WRITE の実装を変更した．PSO についても同様に変更した．

```
1 #define(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;\ /* 不要な状態を作らないため */
8 }\
9 }
```

図 10 TSO の新しい WRITE マクロの実装
Fig. 10 Implementation of new WRITE macro for TSO.

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

図 11 TSO の SVAR マクロの実装
Fig. 11 Implementation of SVAR macro for TSO.

5.2 LTL 式から共有変数を参照するためのマクロ

本節では，LTL 式から共有変数の値を参照するための GSVAR と SVAR の実装と使用方法を述べる．

5.2.1 実装

SC では特別な工夫は必要なく，通常の Promela の方法で共有変数の値を返すように実装する．具体的には，GSVAR は引数の共有変数の値を返すように実装し，SVAR は第一引数のプロセスによらず第二引数の共有変数の値を返すように実装する．

TSO では，各プロセスが書き込んだ内容がプロセス毎のストアバッファに溜まっていることに注意する．GSVAR は，ストアバッファの内容に関係なく引数の共有変数の値を返すようにする．SVAR は，**図 11** のように，第一引数のプロセスのプロセス ID に対応するストアバッファに第二引数の共有変数を対象とするストアが格納されていなければ共有変数の値を，格納されていればそのプロセスのプロセス ID に対応する共有変数のコピーの値を返すようにする．なお，Promela で特定のプロセスのプロセス ID はプロセス名:_pid のように参照できる．

PSO も TSO と同様の方法で定義する．

5.2.2 使用方法

LTL 式中で共有変数の値を参照する際，参照したい値は共有変数が持つ値なのか，いずれかのプロセスから観測される共有変数の値なのかに注意する必要がある．TSO や PSO では，これらは異なることがあるからである．例えば図 2 のモデルで，共有変数 want0 が 1 になっていてもいつかは 0 に戻って，14 行目にあるプロセス t1 の

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

のブロックが解除されることを検査したいとする．その時

は，プロセス `t1` からどう観測されるのかが重要なので，`SVAR` を使って，

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

とするのがよい．また，図 2 のモデルには現れていないプロセスがあり，そのプロセスから共有変数 `want0` の値の変化がどのように観測されるかに興味がある時は，次の例のように `GSVAR` を使って共有変数の値を直接調べることができる．

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

ところで，Promela では，初期値を明示しない場合は変数の初期値は 0 になる．本研究では，共有変数の初期値を設定するために `INIT` を提供するが，`INIT` で初期値を設定するまでは共有変数の値は 0 のままである．そのため，例えば`<> (GSVAR(x) == 0)` などの LTL 式は初期状態で成り立ってしまい，正しく検査できない．しかし，初期化が完了したことを明示的に示すように，ユーザがユーザモデルと LTL 式を変更すればこのような性質も検査できる．例えば，モデルの初期化が完了したことを示すフラグを通常のグローバル変数として導入し，init プロセス内の共有変数の初期化が終わった位置でフラグを操作する方法がある．フラグとしてグローバル変数 `flag` を導入し，以下のように操作する．

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

この初期化が終わって `flag` がセットされている状態を m_0 と書くことにする．このような状態は一状態しかない．さらに，LTL 式 φ を以下のように変形する．

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

以下では，元々の LTL 式 φ と変換後の LTL 式 $\Box(\texttt{flag} \to \varphi)$ が同じ性質を表現することを証明する．

クリプキモデル $\mathfrak{M}$ を $\mathfrak{M} = \langle M, R, val\rangle$,　$m_0 \in M$, $val(\texttt{flag}) = \varnothing$ と仮定する．$\mathfrak{M}' = \langle M', R', val'\rangle$ を

$$m'_0 \notin M \qquad M' = M \cup \{m'_0\}$$

$$R' = R \cup \{\langle m'_0, m_0\rangle\} \qquad val'(\texttt{flag}) = \{m_0\}$$

$$\text{任意の } p \neq \texttt{flag} \text{ について } val'(p) \setminus \{m'_0\} = val(p)$$

を満たすクリプキモデルとする．

補題 1. φ が `flag` を含まないとき，$\mathfrak{M}, \pi \vDash \varphi$ であることと $\mathfrak{M}', \pi \vDash \varphi$ であることは同値である．

証明．φ に関する帰納法による．□

命題 2. φ が `flag` を含まないとき，$\mathfrak{M}, m_0 \vDash \varphi$ であることと $\mathfrak{M}', m'_0 \vDash \Box(\texttt{flag} \to \varphi)$ であることは同値である．

証明．$\mathfrak{M}, m_0 \vDash \varphi$ と仮定する．π を $\mathfrak{M}'$ 上の m'_0 を始点とする経路とする．π は $m'_0 m_0$ というプレフィクスを持つので，π^1 は $\mathfrak{M}$ 上の m_0 を始点とする経路である．ゆえに，$\mathfrak{M}, \pi^1 \vDash \varphi$ である．

定義より，

$$\begin{aligned}
&\mathfrak{M}', \pi \vDash \Box(\texttt{flag} \to \varphi) \\
\iff& \text{任意の } i \in \mathbb{N} \text{ について } \mathfrak{M}', \pi^i \vDash \texttt{flag} \to \varphi \\
\iff& \mathfrak{M}', \pi^1 \vDash \texttt{flag} \to \varphi \text{ かつ} \\
& \quad \text{任意の } i \neq 1 \text{ について } \mathfrak{M}', \pi^i \vDash \texttt{flag} \to \varphi \\
\iff& \mathfrak{M}', \pi^1 \vDash \varphi
\end{aligned}$$

であり，補題 1 より，$\mathfrak{M}', \pi^1 \vDash \varphi$ であることと $\mathfrak{M}, \pi^1 \vDash \varphi$ であることは同値なので，$\mathfrak{M}', m'_0 \vDash \Box(\texttt{flag} \to \varphi)$ である．

$\mathfrak{M}, m_0 \nvDash \varphi$ と仮定する．定義より，$\mathfrak{M}$ 上の m_0 を始点とする経路 π が存在して $\mathfrak{M}, \pi \nvDash \varphi$ である．補題 1 より，$\mathfrak{M}', \pi \nvDash \varphi$ である．$m_0 \in val'(\texttt{flag})$ なので，$\mathfrak{M}', \pi \nvDash \texttt{flag} \to \varphi$ である．π にプレフィックス m'_0 を追加した列（$m'_0\pi$ と書く）は $\mathfrak{M}'$ 上の経路なので，$\mathfrak{M}', m'_0\pi \nvDash \Box(\texttt{flag} \to \varphi)$ であり，ゆえに，$\mathfrak{M}', m'_0 \nvDash \Box(\texttt{flag} \to \varphi)$ である．□

LTL 式内で共有変数以外のグローバル変数やローカル変数，ラベルなどのライブラリが管理しないものについては，2.3 節で述べたような通常の Promela と同じ方法で参照する．

5.3 モデル化の正しさ

本ライブラリでは，ストアバッファと共有変数のコピーを用いて TSO と PSO をモデル化した．これはメモリモデルの操作的なモデル化では一般的であり，既存研究でも同様のモデル化で説明されている [10], [13]．本節では，Travkin らの TSO と PSO のメモリモデルの説明 [13] との対応を示すことで，本研究で改良したライブラリで用いたモデル化の正しさを説明する．

Travkin らは TSO と PSO のメモリモデルを，各プロセスがストアバッファを持つモデルとして説明している．Travkin らのストアバッファは，共有メモリや他のプロセスのストアバッファと一貫性がないキャッシュとしても働く．プロセスからの書き込み（*write* 操作）はストアバッファでキャッシュされ，ストアバッファを時々フラッシュする（*flush* 操作）ことで共有メモリに反映される．なお，*flush* 操作はストアバッファの最も古い書き込み一つだけを共有メモリに反映する操作になっている．プロセスからの読み込み（*read* 操作）では，対象の変数の値がストアバッファにキャッシュされていない時だけ共有メモリから読み込む．フェンス（*fence* 操作）は *flush* 操作を強制してストアバッファを空にする操作として説明されている．

本ライブラリでは，Travkin らのストアバッファの，共有メモリへの書き込の反映を遅延させる機能とキャッシュ

の機能を別の仕組みで実現している．前者は本論文では Promela のチャネル（本論文ではこれをストアバッファと呼んでいる）とメモリプロセスによって実現している．後者は共有変数のコピーにより実現している．本ライブラリではメモリプロセスが書き込みを反映させるのに対して，Travkin らの説明ではユーザプロセスが自ら非決定的に *flush* 操作を行うという違いがある．しかし，一つの書き込みを共有メモリに反映させる操作は不可分に行われるため，どのプロセスが行ってもモデル検査の結果に影響せず，両者に本質的な違いはない．

Travkin らの *write* 操作と *fence* 操作は本ライブラリの `WRITE` マクロ と `FENCE` マクロに対応している．*read* 操作は改良前の本ライブラリの `READ` マクロと同じように，ローカル変数を引数にとって共有変数の値をレジスタにコピーする意味論になっている．したがって，改良後の `READ` マクロと厳密には対応しないが，改良後の `READ` マクロ を使って

a = `READ`(x);

のようにローカル変数（a）に共有変数（x）の値を読み込むことはできる．このような使い方をすることで，本ライブラリは Travkin らの TSO と PSO の説明に沿った振舞いをする．

それ以外の使い方で `READ` マクロを使う場合も Travkin らの TSO や PSO の説明の範囲を超えた振舞いはしない．例えば，

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

のように `WRITE` マクロの引数として `READ` マクロを使った場合は，5.1 章で示したように，`WRITE` マクロの中で `READ` マクロ の結果を一時的なローカル変数に格納してそれを書き込む操作になるので，

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

と同じ振舞いになり Travkin らの TSO や PSO の範囲を超えない．また，

```
(READ(x) == 0)
```

のようにガードとして使う場合は，

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

に置換えたモデルの実行トレースのうち，`READ(x)` の結果が 0 だったものだけを検査することになり，やはり Travkin らの TSO と PSO の範囲を超えない．以上より，本ライブラリのモデル化は Travkin らの TSO や PSO の説明に沿っている．

6. 実験

本ライブラリが提供するマクロを使用して， ライブラリが正しく実装できているかどうかの調査と，共有変数を参照する LTL 式を使用した検査が正しくできるかどうかの調査，その性能の評価を行った．

ライブラリが正しく実装できているかどうかは文献 [15] と同じ方法で調査した．具体的には x86-TSO に従ったメモリモデルになっているかどうかをテストするためのテストプログラム集 (x86-TSO リトマステスト) [11] を用い調査した．x86-TSO リトマステストにはプログラムと x86-TSO のもとで起こり得る結果や起こり得ない結果が記されている．本ライブラリの TSO のモデルは x86-TSO リトマステストの期待される結果に合致するかを，PSO のモデルは既存のメモリモデルを考慮したモデル検査器である McSPIN [3] での実行結果と比較することで調査した．その結果，全てのテストにおいて期待する結果になった．

以下では LTL 式を仕様した検査が正しくできるかどうかの調査と性能評価の結果を示す．

6.1 実験方法

共有変数を参照する LTL 式を使用した検査が正しくできるかどうかの調査と性能の評価には，文献 [16] と文献 [9] で紹介されているモデルのうちの一部を使用した．文献 [16] からは， **表 1** に示す，LTL 式を使用しかつ検査結果が述べられているモデルを使用した．ただし，本ライブラリは数値型の共有変数のみを扱うため，本ライブラリでは扱わないチャネル型の共有変数を持つ一つのモデルは除外した．「モデル名」の項目はモデルの文献 [16] 中で記載されている箇所を表す．表 1 のモデルのうち，練習 2.1 と問題 3.7 は活性と安全性を LTL 式で検査する方法が記載されていたため，それぞれを検査した．文献 [9] からは， **表 2** に示す，2 プロセスについての相互排除アルゴリズムのモデル例を使用した．「モデル名」の項目はモデルで使用されている相互排除アルゴリズム名を表す．文献 [9] では，それぞれのモデルの TSO と PSO に従った実行を検査する際に，相互排除を成り立たせるためにフェンス命令を挿入するべき位置が明示されている．つまり， 表 2 に示すモデルは，安全性が成り立つモデルである． これらのモデルには，共有変数の読み込みや，共有変数を参照するガード，共有変数の初期値を設定するモデルが含まれている．

LTL 式内から共有変数を参照するマクロが正しい動作をするかどうかは，文献 [16] で述べられている検査結果との比較と，文献 [9] のモデルで活性と安全性の性質が成り立つかどうかで判断した．3.5.2 節，問題 3.1，問題 3.7，Dijkstra は共有変数に 0 以外の初期値を設定する必要のあるモデルのため，これらのモデルを検査することで，`INIT` の正しさを調査する．性能の評価は，使用した総メモリ使用量（一つの状態を表すのに使用したバイト数と状態数を掛け合わせたもの）と実行時間の観点から行った．

表 1 のモデルと 表 2 のモデルについて，LTL 式を使用して検査した．本ライブラリを使用しない検査では， 表 1 のモデルは文献 [16] の記載されているそのままのモデルを，表 2 のモデルは通常の Promela で記述したモデルを検査した．本ライブラリを使用した検査では，表 1 と 表 2

のモデルを，本ライブラリを使用するように変更したモデルを検査した．

表 1 と 表 2 の「モデル名」の項目のそれぞれの名称の後に，活性を検査した場合は (L) を，安全性を検査した場合は (S) を記した．表 1 のモデルについては，文献 [16] に記載されていた LTL 式を使用して検査を行った．表 2 のモデルについては，クリティカルセクションを表すラベル CS を追記して LTL 式による検査を行った．例えば，プロセス P の活性を検査する際，P が共有変数 F を 1 にしてクリティカルセクションに入ることを表明するならば，以下のような式を記述した．

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

また，活性を検査する際には，-f オプションを付けて弱い公平性の下でモデル検査を行った．

このとき，ライブラリのパラメタのうち PROCSIZE と VARSIZE については，各モデルのユーザプロセスの数やグローバル変数の数に合わせて定義した．モデルの中には，書き込みを含みかつ繰り返しの回数が一定ではないユーザプロセスを含むものがあるため，BUFFSIZE はどのモデルでも 5 と定義した．さらに，表 2 のモデルを TSO と PSO で検査する際は，文献 [9] に載せられているモデル例の通りに，それぞれのメモリモデルで相互排除が成り立つようにフェンス命令を挿入した．

実験に使った環境は，Ubuntu 16.04.2 LTS，Intel Core i7-6700K 4.00GHz である．

6.2 実験結果

表 1 の SC については，どのモデルを検査した結果も文献 [16] の結果と一致した．TSO と PSO については，活性と安全性を検査した結果が文献 [16] と異なっていたが，これらのモデルは実行順序が緩和されているメモリモデルでは相互排除が成り立たないアルゴリズムのモデルのため妥当であると考えられる．3.5.2 節のモデルはユーザプロセスの数が一つのためメモリモデルの影響を受けないが，TSO と PSO で検査した結果が文献 [16] と一致したため，TSO と PSO のマクロの実装が単一のユーザプロセスしか持たないモデルでも正しく動作することが確認できた．問題 3.1 もユーザプロセスの数が一つのモデルであり，TSO で文献 [16] の結果と一致することを確認した．しかし，PSO では状態数が大きくなり過ぎたために 47GB のメモリを使用した時点で検査を打ち切った．

表 2 のモデルの活性については，SC で検査した結果は通常の Promela で作成したモデルを検査した結果と一致した．安全性については，Peterson のモデル例を PSO で検査した場合に満たされなかった．文献 [9] の記述を確認したところ，実験の際に PSO で相互排除を成り立たせるためにフェンス命令を挿入したことがわかる[*5]が，モデル例には PSO で検査する際にフェンス命令を挿入する箇所が示されていなかった．そこで，PSO で相互排除が成り立つように Peterson のモデル例にフェンス命令を挿入したモデルを検査し，その結果を Peterson∗ に示した．Peterson∗ では PSO でも安全性が満たされ，相互排除が成り立つことを確認した．

以上より，本ライブラリが提供するマクロを使用して，共有変数を参照する LTL 式を使用した検査が正しくできることが示せた．

表 1 の「メモリ」と「実行時間」の項目に，それぞれのモデルを検査した際に要した総メモリ使用量（KB）と実行時間（秒）を示す．ただし，これらの結果は SPIN が反例を検出するか，反例を検出せずに検査を終了するまでに要した総メモリ使用量と実行時間である．

どのモデルのどのメモリモデルも，文献 [16] や文献 [9] のモデル以上のメモリと実行時間を要した．

例えば，問題 2.1 のモデルの活性を PSO で検査した際は，通常の Promela で記述された文献 [16] のモデルの 1000 倍以上のメモリを要した．これは，表 1 の問題 3.1 以外の他のモデルでは一つのユーザプロセスが一つの共有変数に対してのみ書き込みを行っていたのに対し，問題 2.1 のモデルでは二つの共有変数に対して書き込みを行ったことが原因だと考えられる．PSO では，各ユーザプロセスは共有変数毎のストアバッファを持っているため，一つのユーザプロセスが複数の共有変数を操作するモデルでは状態数が大きく増えることになる．対して，一つのユーザプロセスが一つの共有変数のみを操作するモデルでは，各ユーザプロセスの持つストアバッファのうち一つのストアバッファしか使用されないため，TSO と同程度の状態数になる．

問題 3.1 のモデルについては，PSO では状態数が大きくなり過ぎたために 47GB のメモリを使用した時点で検査を打ち切った．その原因は，問題 3.1 のモデルは一つのユーザプロセスが多数の共有変数へ書き込みを行ったことだと考えられる．前述したように，PSO ではユーザプロセス毎かつ共有変数毎のストアバッファを持つため，一つのユーザプロセスが複数の共有変数へ書き込むモデルでは TSO よりも状態数が増える．問題 3.1 は一つのユーザプロセスが 6 つの共有変数へ書き込むモデルである．二つの共有変数へ書き込む問題 2.1 のモデルでも通常の Promela のモデルの 1000 倍以上のメモリを要したことを考えると，問題 3.1 のモデルの検査には膨大なメモリと実行時間がかかることが予想され，実際にそのような結果が得られた．なお，問題 3.1 のモデルのように，ユーザプロセスが一つのモデルでは，どのメモリモデルでも通常の Promela で記述したモデルと実行結果は変わらないはずのため，本ライブラリ

*5 p. 161

表 1 モデル検査の教科書のモデルをモデル検査した結果
Table 1 Results of model checking for models in a textbook on model checking.

モデル名	結果							
	文献		SC		TSO		PSO	
	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）
3.5.2 節	46.5	0	129.6	0	4210.4	0.02	4991.5	0.02
練習 2.2	0.2	0	0.3	0.01	24.9	0.07	26.1	0.07
問題 2.1(L)	1.7	0	5.0	0.11	130.6	0.02	1914.0	0.02
問題 2.1(S)	2.7	0	2.5	0.02	5.3	0.13	8.8	0.13
問題 3.1	0.8	0	1.8	0	1477.4	0.01	>47GB	>42.6
問題 3.3	1.9	0	2.3	0	150.0	0.07	146.3	0.07
問題 3.7(L)	1.5	0	2.0	0	251.6	0.13	176.4	0.13
問題 3.7(S)	0.8	0	1.1	0.03	36.9	0.06	25.9	0.06

表 2 メモリモデルを考慮したモデルをモデル検査した結果
Table 2 Results of model checking for intended to run correctly under the memory models.

モデル名	結果							
	文献		SC		TSO		PSO	
	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）	メモリ（KB）	実行時間（秒）
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	0.1	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

を使用せずに検査できる．

以上より，本ライブラリは，一つのユーザプロセスが多くの共有変数へ書き込むモデルでは PSO に従った実行を検査する際に状態数が爆発する，という特徴を有することがわかる．なお，本ライブラリと同様に，ストアバッファを用いて共有変数への書き込みを遅延させるアイデアによって PSO を実現している手法 [13], [14] も同じ特徴を有していると考えられる．

一方，問題 3.7 のモデルを検査した際に使用したメモリは，TSO より PSO の方が少なくなっていた．このモデルでは一つの状態を表すのに必要なバイト数が TSO より PSO の方が少なくなっていた．これは，TSO の実装で必要な各プロセス毎のカウンタが PSO では不要なことや，TSO ではストアバッファにメモリアドレスと値の組を格納しているのに対して PSO では値のみを格納していることが原因である．ユーザモデルによっては PSO のモデルのサイズが TSO よりも小さくなることがある．表 2 の Lamport Fast のモデルを検査した場合も，TSO より PSO の方が少ないメモリ使用量で検査を終えているが，TSO よりも PSO の方が一つの状態を表すのに必要なバイト数が大きく，状態数が少なかったことから，PSO では TSO で緩和されていないストア → ストアの実行順序が緩和されていることによって，早期に活性が成り立たないことを検出したと考えられる．

実行時間については，ほとんどのモデルで SPIN で計測できる最短の 0.01 秒より短く，問題 3.1 以外の最も長いモデルで 0.13 秒であった．

7. 関連研究

本研究以前にも，SPIN でメモリモデルに従った実行を検査する手法が提案されている [13], [14], [17].

Wehrheim らは，C/C++のコードを LLVM の IR コードを介して TSO，PSO に従った動作をする Promela コードに変換する手法を提案している [13], [14]．この手法は，本研究の手法と同じく，ストアバッファによって TSO と PSO で起こり得るメモリアクセス命令の実行順序の入れ

替わりを表現するアイデアによるものである．Wehrheim らの手法では通常の Promela と同様に，LTL 式と assert 文によって検査した性質を記述する．LTL 式内からどのグローバル変数の値でも参照できるが，参照できるのは実際にメモリに反映された値のみで，本研究の手法が提供する SVAR 命令のような，ある特定のプロセスが観測できるグローバル変数の値を参照することができない．本研究の手法では，実際にメモリに反映された値だけでなく，ある特定のプロセスが観測できるグローバル変数の値も参照できる．また，Wehrheim らの手法によって生成される Promela コードは可読性が低いにも関わらず，検査したい性質を記述するためにはそのコードを読む必要がある．本研究の手法では，メモリモデルを考慮しないモデル内の共有変数に対するメモリアクセス命令を，ライブラリが提供するメモリアクセス命令にほとんど置き換えるだけで TSO と PSO に従った実行を検査できる．そのため，コードの可読性が高く，検査したい性質を容易に記述できる．

Wehrheim らは，SC，TSO，PSO に従った実行を検査できる Promela で記述されたライブラリを，[13], [14] で実験の比較対象として使用している．このライブラリは，本研究と同じく，ストアバッファを用いたアイデアによって実装されている．しかし，Wehrheim らのライブラリでは，グローバル変数がユーザから隠されているため，我々が以前に開発したライブラリと同様にグローバル変数を参照した LTL 式を記述できない．改良したライブラリでは，ユーザが共有変数（複数のプロセスがアクセスする場合のあるグローバル変数）にアクセスできるように，専用のマクロを提供することで，Wehrheim らのライブラリの欠点を克服している．

Tomasco らは，メモリモデルに従った動作をする API を設計し，CBMC[7] や Nidhugg[1] をバックエンドとして利用して，メモリモデルに従ったモデル検査を提供している [12]．しかし，Tomasco らの手法では LTL 式による検査をサポートしていない．

Abdulla らは，ステートレスモデル検査によるメモリモデルを扱えるモデル検査器を提案している [1], [2]．Abdulla のモデル検査器は C で記述されたマルチスレッドプログラムのみを対象としているが，本研究の手法では Promela でモデル化することで C 以外のマルチスレッドプログラムも検査できる．

8. おわりに

本論文では，我々が開発してきた，SPIN で弱いメモリモデルに従った実行を検査できるようにするためのライブラリを改良した．具体的には，まずガードからも共有変数を参照できるように改良した．さらに，共有変数の初期値を設定できるように改良した．最後に，検査したい性質を LTL 式で記述する際に共有変数を参照できるように改良した．

改良したライブラリを実装し，実験を行った結果，改良したライブラリが提供するマクロの動作が正しいことを確認した．性能の評価を行った結果，本ライブラリは一つのユーザプロセスが多くの変数へ書き込むモデルでは PSO に従った実行を検査する際に状態数が爆発する，という特徴を有することが示された．実行時間については，検査を打ち切ったモデル以外の実験では，現代の計算機で十分実行可能な実行時間で検査を完了した．

今後の課題としては，まず，本ライブラリと関連研究として紹介した Wehrheim らのライブラリ [13], [14] で TSO と PSO の動作に違いがあるかどうかの調査が挙げられる．他にも，これまでに扱わなかったメモリモデルへの対応や，キャッシュコヒーレンシを考慮したモデルの開発が挙げられる．また，ライブラリのパラメタのうちプロセスの数と共有変数の数についてはユーザの責任としているが，これらを検査対象のプログラムから自動的に抽出する仕組みを開発することは，ライブラリの使いやすさのためには急務である． 本ライブラリでは書き込みがストアバッファから溢れるとブロックするが，ストアバッファから溢れるとエラーとする手法も存在する [13], [14]．エラーにする方がユーザは正しく検査できていないことを認識しやすいが，本ライブラリでは有界モデル検査で使用することも考えてブロックするように実装した．今後，エラーにする機能も追加して拡張したい． さらに，本ライブラリでは CAS 命令のような CPU 毎にセマンティクスが異なる命令を提供していない．本ライブラリが提供するマクロと Promela の機能を組み合わせることでユーザが定義できる命令もあるがそうでないものもあるため，代表的な命令が定義できるように改良することも今後の課題である．

謝辞 本研究の一部は，JSPS 科研費 16K21335 と 16K00103 の助成を受けたものです．

参考文献

[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. of 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. of 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] 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).

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

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

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

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

[10] 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).

[11] 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).

[12] 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. of FMCAD* (2016).

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

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

[15] 松元稿如，鵜川始陽，安部達也：メモリモデルを考慮したメモリアクセス命令を提供する SPIN 用ライブラリ，ソフトウェア工学の基礎，Vol. 23, pp. 63–72 (2016).

[16] 産業技術総合研究所システム検証研究センター：モデル検査 上級編－実践のための三つの技法－，株式会社近代科学社 (2010).

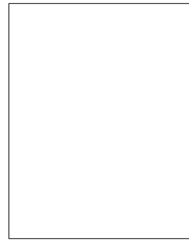
[17] 加藤寿和，一場利幸，本田晋也，高田広章：ハードウェアの振舞いを考慮したスピンロックのモデル検査，研究報告組込みシステム（EMB），Vol. 2011, No. 2, pp. 1–8 (2011).

松元 稿如

1994 年生．2017 年高知工科大学情報学群卒業．2016 年 IEEE Computer Society Japan Chapter FOSE Young Researcher Award 受賞．

鵜川 始陽 （正会員）

1978 年生．2000 年京都大学工学部情報学科卒業．2002 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年同専攻博士後期課程修了．同年京都大学大学院情報学研究科特任助手．2008 年電気通信大学助教．2014 年より高知工科大学准教授．博士 (情報学)．プログラミング言語とその処理系に関する研究に従事．情報処理学会 2012 年度山下記念研究賞受賞．



安部 達也 （正会員）

1979 年生．2002 年京都大学理学部卒業．2007 年東京大学大学院情報理工学系研究科博士後期課程退学．同年産業技術総合研究所システム検証研究センターテクニカルスタッフ．2008 年同センター特別研究員．2009 年京都大学学術情報メディアセンター特定助教．2011 年理化学研究所計算科学研究機構特別研究員．2013 年同機構研究員．2015 年千葉工業大学人工知能・ソフトウェア技術研究センター上席研究員，現在に至る．博士 (情報理工学)．プログラミング言語の理論と実装，プログラム検証の研究に従事．