# SPINによるメモリ順序を考慮したソフトウェアモデル検査の反例可視化

松元 稿如[1,a)]　鵜川 始陽[1,b)]

**概要**：現代のプロセッサでは，メモリアクセス命令がプログラムに記述された順序（プログラム順序）と，それがメモリに反映される順序（メモリ順序）が異なることがある．そのため，並行プログラムの検査では，プログラム順序だけでなくメモリ順序を考慮する必要がある．これまでに我々は，モデル検査器 SPIN を使って，メモリ順序を考慮したモデル検査を行うためのライブラリ MMLib を開発した．SPIN は，命令型のモデル記述言語 Promela で記述したプログラムのモデルを入力とする．モデル検査によりバグが見つかった時は，Promela で記述したモデル上のエラーに至る実行トレースである反例を読み解くことになる．しかし，MMLib を使用したモデルの反例では，メモリ順序が MMLib が内部的に使う変数の値からしか分からず，読み解くのが難しい．そこで我々は，MMLib を使用したモデルの反例を可視化するソフトウェアを開発した．このソフトウェアは反例に沿ってモデルの実行を再現する．その時に， メモリに反映されていないメモリアクセス命令をハイライト表示し，メモリ順序が分かるようにする．¿¿¿¿¿¿ master2

**キーワード**：SPIN，メモリモデル，モデル検査，デバッグ，可視化

# Visualization of Counterexamples of Memory Model-Aware Software Model Checking Using SPIN

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

***Abstract:*** In modern processors, the memory order, which is the order of memory access operations reflected to the memory, is not necessarily the same as the program order, which is the order of memory access instructions listed in the program. Thus, when we verify concurrent programs, we have to take the memory order into account. In our previous work, we developed a model checking library, MMLib, which helps us conduct model checking with taking the memory order into account using the SPIN model checker. The input of SPIN is a model of the program written in Promela, an imperative style modeling language. When SPIN reports an error, the user of SPIN has to understand the counterexample, which is the execution trace on the model written in Promela resulting in the error. However, counterexamples of models written using MMLib are difficult to understand because we need to interpret the values of variables used internally in MMLib. Therefore, we developed a software to visualize counterexamples of models using MMLib. The software visualizes the execution of the model step by step along a counterexample. In the step execution, it highlights memory access instructions that have been executed but yet to be reflected to memory to visualize the memory order.

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

## 1. はじめに

並行プログラムのモデル検査では，メモリアクセス命令がプログラムに記述された順序（プログラム順序）だけでなく，それがメモリに反映される順序（メモリ順序）も考

[1] 高知工科大学
Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] matsumoto@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp

慮する必要がある．現代のプロセッサでは，プログラム順序とメモリ順序が異なる場合がある．その結果，プログラム順序通りにメモリアクセス命令がメモリに反映されれば正しく動作するプログラムでも，メモリ順序によってはエラーが発生する場合がある．これは，命令を実行したのとは別のスレッドからは，命令がメモリ順序で実行されたように見えるからである．命令がどんなメモリ順序で実行されることがあるか定義したものを**メモリモデル**と呼ぶ．

これまでに我々は，モデル検査器 SPIN を使いメモリモデルを考慮してプログラムをモデル検査するためのライブラリ MMLib を開発した [1], [2]．MMLib は，SPIN[3] のモデル記述言語 Promela で記述されたモデルの部品である．プログラムのモデルと MMLib を結合して SPIN に入力することで，メモリモデルを考慮したモデル検査を行える．

MMLib は，プログラム順序とメモリ順序が異なることに起因するバグの発見とデバッグに用いることを想定している．このユースケースでは，プログラム順序での実行ではエラーが起きないことを他のツールなどを使って確かめた後に，MMLib を用いたモデル検査を行う．そのため，本研究でも，MMLib を使ったモデル検査の対象は，プログラム順序の実行ではエラーが起きないことを仮定する．

SPIN のモデル記述言語 Promela は命令型の言語である．SPIN は，Promela で記述されたモデル中の各スレッド[*1]をインタリーブしながら 1 文ずつ順に実行した時に，エラーに至る可能性があるかどうかを検査する．エラーに至る可能性はバグである．バグが見つかった時は，モデル上のエラーに至る実行トレースである反例を出力する．

モデル検査を利用してプログラムをデバッグする場合，ユーザは反例からバグの原因を読み解く必要がある．しかし，MMLib を使用したモデル検査の反例は次の二つの理由から読み解くのが難しい．一つ目は，反例中に MMLib の内部的な実行ステップが含まれるためである．MMLib はプログラムのモデルと同じく Promela で記述されているため，SPIN は MMLib の内部のモデルの実行とプログラムのモデルの実行とを区別せず反例を出力する．そのため，ユーザが反例の実行ステップとプログラムのモデル上の行との対応を取るのは難しい．二つ目は，メモリアクセス命令がどのようなメモリ順序で実行されたのかは，MMLib の内部的な変数の値から読み解く必要があるためである．この二つの理由のうち，前者は MMLib に限らず，同様のマクロを使ったモデル検査ライブラリにも共通する．後者は，メモリ順序を考慮したデバッグ特有である．本論文では後者に焦点を絞り，メモリ順序を可視化する方法を提案する．

我々の経験では，メモリモデルを考慮したモデル検査によって，プログラムのバグを理解しデバッグするには，各実行ステップについて以下の情報が必要であった [4], [5]．

**(1)** グローバル変数やローカル変数のメモリ上の値と，各スレッドから見える値．

**(2)** 各スレッドが実行したメモリアクセス命令のうち，メモリに反映されていないメモリアクセス命令．

そこで我々は，MMLib を使用したモデル検査の反例を可視化し，上記の情報をユーザに提示できるソフトウェアを開発する． 本ソフトウェアは，反例に沿ってモデルの実行を再現し，モデル上に視覚的に上記の情報を提示する．本反例可視化ソフトウェアは，一般的な命令型言語のデバッガと同様に，モデルの文単位や行単位でステップ実行の様子を表示する．さらに，まだメモリに反映されていないメモリアクセス命令や，プログラム順序とメモリ順序が異なったメモリアクセス命令をハイライト表示する．

本論文では，まず 2 章で例を用いて MMLib を使ってデバッグする際の問題点を挙げる．次に，**??**章で MMLib の内部構造を説明する．その後，3 章で本反例可視化ソフトウェアの設計を説明 する．4 章で本反例可視化ソフトウェアを使ったデバッグとバグの理解の事例を示し，5 章で既存の反例可視化ソフトウェアと比較する．さらに，6 章で関連研究について言及し，7 章で本論文のまとめと今後の課題を述べる．

## 2. MMLib を使ったデバッグの問題

### 2.1 ピーターソンの相互排除アルゴリズム

**図 1** はピーターソンの相互排除アルゴリズム [6] のプログラムである．ピーターソンの相互排除アルゴリズムは，メモリアクセス命令がプログラム順序通りにメモリに反映されることを仮定している．その仮定が成り立たないメモリモデルでは排他制御ができない．

関数 `T0` と `T1` は異なるスレッドで実行されることを想定しており，`Critical Section` というコメントが排他制御の対象となるクリティカルセクションである．共有変数 `want0` と `want1`（以下，まとめて `want` と書く）は，それぞれ `T0` と `T1` を実行するスレッドがクリティカルセクションに入りたいという表明である．クリティカルセクションに入りたいスレッドはそのスレッドの `want` を 1 にする．その後，他方のスレッドの `want` を読んで，0 であればクリティカルセクションに入れる．他方の `want` を読むのは自身の `want` を 1 にした後なので，両方のスレッドが同時にクリティカルセクションに入ることはない．しかし，これだけでは両方のスレッドが同時に自身の `want` を 1 にしてデッドロックする可能性がある．`turn` はそれを解決するための変数である．

### 2.2 MMLib を使った Promela モデル

図 1 のプログラムのモデルを MMLib を使って Promela で記述すると**図 2** のようになる．Promela ではスレッド

[*1] Promela ではプロセスと呼ぶ．

の定義に proctype というキーワードを使う．図 1 の関数 T0 と T1 は図 2 の T0 と T1 に対応している．図 2 の 38 行目にある ltl キーワードの行は，両方のスレッドが同時にクリティカルセクションに入ることはないという仕様を線形時相論理式 (LTL 式 ) で記述したものである．SPIN はこの仕様を満たさない実行トレースを探す．

Promela のモデル中で，複数のスレッドからアクセスされる変数（**共有変数**）にアクセスするには MMLib を使う．MMLib は共有変数を管理しており，共有変数にアクセスするマクロを提供している．つまり，共有メモリとメモリアクセス命令は，MMLib では，共有変数と**共有変数アクセス命令**としてモデル化されている．MMLib は，共有変数アクセス命令として共有変数 $x$ から値を読み出す READ($x$) や，$x$ に値 $v$ を書き込む WRITE($x$, $v$) に加えて，プログラム順序で先行する命令と後続の命令が反映される順序を強制するためのフェンス命令のマクロ FENCE() を提供している．

MMLib では共有変数は整数で識別するため，WRITE の第一引数には共有変数を識別する整数を渡す必要がある．図 2 では，Promela で C 言語のマクロが使用できることを利用して，5 行目から 7 行目のように変数名を表すマクロを定義している．これにより，WRITE の第一引数に変数名を渡せるようにしている．例えば，共有変数 turn に 1 を書き込むには，11 行目のように WRITE(turn, 1) と記述する．同様に，turn の値を読み出すには，13 行目のように READ(turn) と記述する．

### 2.3 エラーの原因

図 2 のスレッド T0 がクリティカルセクションに入るためには，12 行目の if 文のいずれかの条件が真になる必要がある．Promela では，条件分岐は if と fi を使って記述する．if 文では，非決定的選択を表す “::” のうち後に続く条件が真である分岐が非決定的に実行される．そのような分岐がなければ，いずれかの条件が真になるまで，そのスレッドの実行がブロックする．

ここでは 14 行目でスレッド T1 がクリティカルセクションに入る表明である want1 を読み出している．スレッド T1 が 21 行目で WRITE(want1, 1) を実行した瞬間に，その結果が共有変数を介してスレッド T0 の 14 行目の READ(want1) で読み出せるようになっていれば，スレッド T0 は if 文でブロックされ，相互排除ができる．しかし，WRITE が共有変数に反映されるのが遅れると，want1 が 0 のままの状態が観測でき，スレッド T0 がクリティカルセクションに入ってしまう．

### 2.4 反例

図 2 のモデルを検査し，SPIN が出力した反例から一部を抜粋したものを**図 3** に示す．ただし，スペースの都合で議論に登場しない部分を省略している．我々の過去の研究 [5] では，この反例を読み解いてバグを理解していたが，それには MMLib の内部構造に関する知識が必要になり，知っていたとしても難しい． 以下では図 3 から，メモリ順序に関連したバグを理解するのに必要になる情報を読み解く方法を示す．

```
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: 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** Program of Peterson’s mutual exclusion program

#### 2.4.1 アクセスしている共有変数の値

READ や WRITE が アクセスする共有変数の値は，MMLib が内部的に使う複数の変数の値に依存する． 例えば，図 3 の反例の 14 行目は，図 2 の 14 行目の READ(want1) に対応している．この READ で読み出される値は，反例の 2 行目から 13 行目の buffer や lcounter，shared_memory に依存する．この例では lcounter[7] が 0 なので，shared_memory[1] の値である 0 が読めている． ここで，配列インデックスの 1 は変数 want1 を識別する整数であり，7 はそこから計算される値である．

#### 2.4.2 未反映の WRITE

共有変数に反映されていない WRITE は，MMLib 内部のキューに保存されている． 図 3 の反例の 16 行目から 20 行目の queue は実行されたが共有変数にはまだ反映されていない WRITE を格納するキューで，19 行目の [1] は T1 が実行した WRITE(want1, 1) を表している．図 2 の 14 行目

```
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)}
```

図 2: MMLib を使用した図 1 のプログラムの Promela モデル

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

で want1 から 0 が読み出される原因が，この WRITE の結果が共有変数に反映されていないためであると理解するには，このように読み解く必要がある．

## 3. 反例可視化ソフトウェア

ユーザが反例を読まずともバグを理解できるようにすることを目的として，反例可視化ソフトウェアを開発する．本反例可視化ソフトウェアの具体的な要件は次の二つである．

(1) WRITE が共有変数に反映されたタイミングを視覚的にユーザに提示する．

(2) 共有変数アクセス命令がアクセスする変数の値を提示する．

本反例可視化ソフトウェアは Eclipse のプラグインとして実装する．

### 3.1 反例可視化ソフトウェアの機能

本反例可視化ソフトウェアは，命令型プログラミング言語の GUI デバッガと同じように，モデル上の実行を 1 ステップずつ表示することで反例を可視化する．反例可視化ソフトウェア上には，同時には一つの実行ステップの情報しか表示しない．反例可視化ソフトウェア上に表示されている実行ステップを**現在の実行ステップ**と呼ぶことにする．

**図 4** は本反例可視化ソフトウェアを用いて図 2 のモデルで見つかったエラーの反例を表示している様子である．本反例可視化ソフトウェアは，操作ビューやモデルビューと，変数表示ビューから構成されている． 変数表示ビューには，共有変数の値を表示する共有変数ビューと，図 4 には表示していないが，ローカル変数の値を表示するローカル変数ビューがある． 操作ビューにはモデルや反例を読み込むための機能や現在の実行ステップを切り替えるためのナビゲーションボタンが配置されている．モデルビューには，ユーザが作成したプログラムのモデルが表示され，その上に現在の実行ステップでの各スレッドの実行位置や，共有変数に未反映の WRITE などの情報が表示される．

本反例可視化ソフトウェアは，上記の要件 (1)，(2) に対応して次の機能を持つ．

(1) 未反映 WRITE 命令ハイライト機能

(2) 各スレッドから観測できる共有変数の値の表示機能

#### 3.1.1 未反映 WRITE 命令ハイライト機能

いずれかのスレッドによって実行はされたが，現在の実行ステップではまだ共有変数に反映されていない WRITE 命令を，モデルビュー上で黄色にハイライトする．MMLib を使ったモデル検査が対象とするのは，プログラム順序での実行ではエラーが起きないプログラムのモデルなので，本反例可視化ソフトウェアを使って理解しようとするバグはメモリ順序に起因するバグである．未反映の WRITE 命令はメモリ順序が確定しておらず，バグを理解するうえで注意する必要がある．

さらに，次のどちらかの条件を満たす WRITE はバグの原因になり得るので，該当の実行ステップでは，モデルビュー上で該当の WRITE を赤色にハイライトする．

条件 a. 共有変数アクセス命令 $M$（WRITE 命令や READ 命令）が共有変数に反映された実行ステップで，プログラム順序で $M$ に先行するが共有変数に未反映で残っている WRITE 命令 $W_0$.

条件 b. 共有変数 $x$ を対象とした READ 命令が共有変数に反映された実行ステップで，他のスレッドが実行し

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

図 3: 図 2 の反例の一部

**Fig. 3** A part of counterexample of Fig.2.

た同じ共有変数 $x$ を対象とする共有変数に未反映の WRITE 命令 $W_1$ のうち，後続のいずれかの共有変数アクセス命令は共有変数に反映されているもの．

条件 a では，この実行ステップで，$M$ と $W_0$ がメモリ順序で入れ替わることが確定するので，バグの原因になり得る．なお，MMLib が扱うメモリモデルでは，READ は実行されると直ちに共有変数に反映される．例えば，図 2 のスレッド T1 によって 25 行目の READ(want0) が実行されたとき，21 行目の WRITE(want1, 1) がまだ共有変数に反映されていなければ，この WRITE を赤色でハイライトする．

条件 b の $W_1$ は，条件 a によって，それ以前の実行ステップで一度ハイライトされているはずである．それを改めてハイライトするのは，実際にメモリ順序が入れ替わったことが他のスレッドに観測されるというイベントが発生するからである．これはバグの原因となり得る．例えば，図 2 のスレッド T0 によって 14 行目の READ(want1) が実行されたとき，スレッド T1 の WRITE(want1, 1) がまだ共有変数に反映されておらず，かつ，READ(want0) は実行されてしまっていれば，この WRITE を赤色でハイライトする．これは，まさに 2.3 節で述べたエラーの原因になっている．

#### 3.1.2 各スレッドから観測できる共有変数の値の表示機能

図 4 の共有変数ビューに示すように，現在の実行ステップの各共有変数の名前と値の表を表示する．表の 1 列目には共有変数名を，2 列目には現在の実行ステップでの共有変数の共有メモリ上での値が表示される．3 列目以降には，各スレッドが現在の実行ステップで観測する共有変数の値が表示される．これは共有メモリ上の値とは異なることがあるが，READ 命令の実装と同じ計算をすることで，MMLib が内部的に用いる変数から再現できる．

表の 1 列目に共有変数名を表示するには，3.2.2 節で後述する共有変数の情報を記述した共有変数情報ファイルで変数名を指定する必要がある．共有変数情報ファイルがなければ，変数名ではなく，変数を識別する整数を表示する．

本反例可視化ソフトウェアには，共有変数だけでなく，ローカル変数を表示するローカル変数ビューもある．ローカル変数はメモリ順序の影響を受けないので，ローカル変数ビューに特別な仕組みは必要ない．

### 3.2 その他の特徴

#### 3.2.1 実行ステップ

SPIN が出力する反例には MMLib の内部的な動作も含まれる．しかし，それらはユーザには必要ないので，ユーザが記述した MMLib を使う Promela のモデルの視点でのイベントのみを実行ステップとして表示する．WRITE や FENCE などのマクロは，不可分に実行される複数の Promela の文で構成されているが，これらはまとめて 1 ステップとする．また，WRITE は専用のプロセスが不可分に実行する複数の Promela の文によって共有変数に反映されるので，これもまとめて 1 ステップとする．

Promela では，モデルが満たすべきでない性質を記述するために never claim という特別なスレッドを利用できる．LTL で記述した仕様も never claim に変換される．never

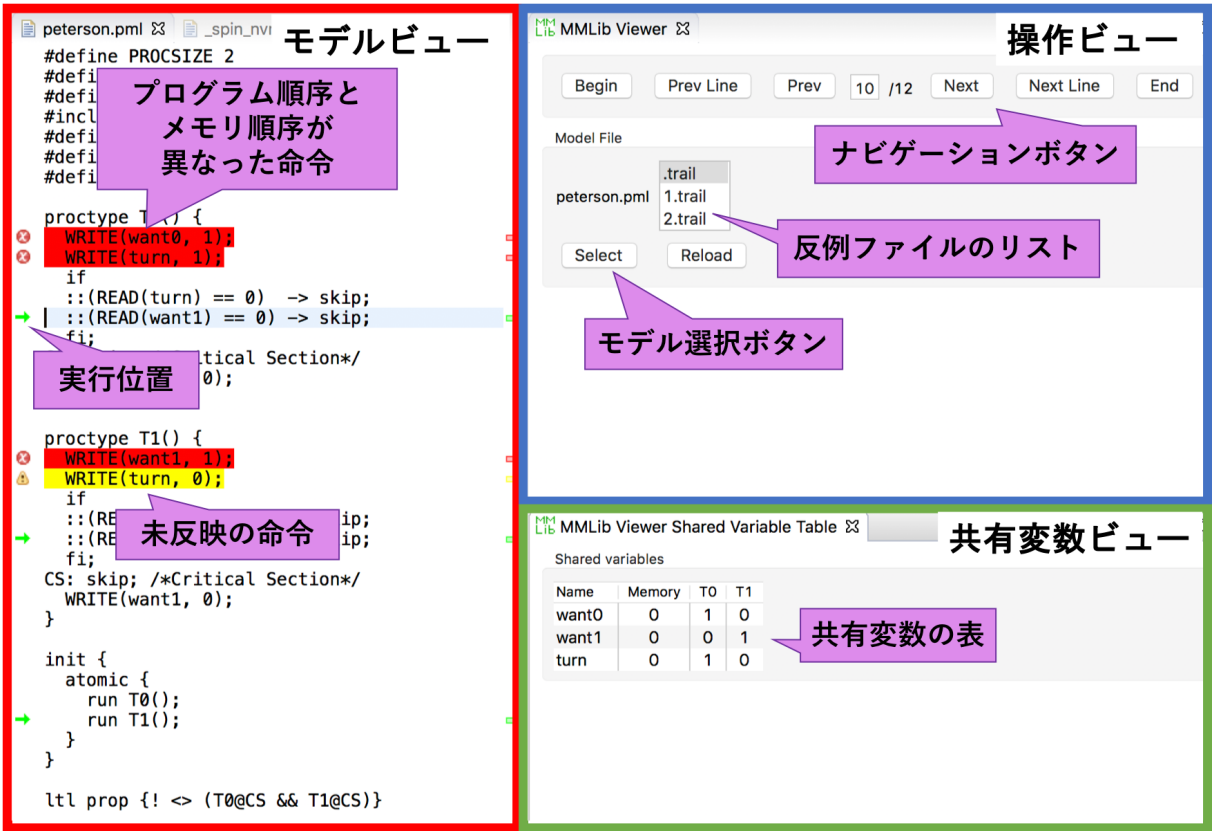



図 4: 反例可視化ソフトウェア
**Fig. 4** Counterexample visualizer.

claim は他のスレッドとは独立して動作し，モデルが満たすべきでない性質を満たすと受理ラベルを持つサイクルに入る．never claim がそのようなサイクルに入ると，SPIN はエラーを検出し反例を出力する[*2]．never claim はプログラムのモデルではないが，その動作はプログラムのモデルをデバッグする上で重要な意味を持つ．そのため，本反例可視化ソフトウェアでは反例中の never claim を実行するステップも 1 ステップとして扱う．

#### 3.2.2 共有変数名

MMLib は共有変数を整数値（以下，変数番号）で識別するが，MMLib を使う Promela のモデル上では通常，記号的な識別子を変数番号で定義して用いる． しかし，このような対応を参照しながら反例を理解するのは手間がかかる． そこで，ユーザがこの対応を共有変数情報ファイルに記述して本反例可視化ソフトウェアに指示することにした．本反例可視化ソフトウェアは共有変数情報ファイルから読み取った名前を使って共有変数ビューを表示する．

## 4. ケーススタディ

本章では，図 1 のピーターソンの相互排除アルゴリズムと，並行ガベージコレクション（GC）の Staccato [7] のアルゴリズムを題材に，本反例可視化ソフトウェアを使ったデバッグとバグの理解の方法を述べる[*3]．

表 1: モデルのサイズ
**Table 1** Sizes of models.

| model | line | thread | shared variable |
|---|---|---|---|
| Peterson | 38 | 2 | 3 |
| Staccato | 174 | 2 | 6 |

**表 1** に，本章で用いたモデルのサイズとして，モデルの行数，スレッド数，および共有変数の数を示す．

### 4.1 バグを理解する方針

次の方針でバグを理解する．まず，短い反例を利用する．一般に，エラーに至る実行トレースがある場合，それは複数ある．SPIN には，最短のステップ数でエラーに至る実行トレースを反例として出力する機能があるので，それを活用する．これは，短かい反例の方がバグを理解しやすいからである．ただし，ここで得られる反例は MMLib の内部的な実行ステップも 1 ステップと数えたときに最短な反

[*2] SPIN は Promela のモデルを Büchi オートマトンに変換して探索する．Büchi オートマトンでは受理状態を含むサイクルに入ると入力を受理したことになる．never claim が受理されるとエラーとして扱われる．

[*3] ピーターソンのアルゴリズムはプログラム順序とは異なるメモリ順序で実行されるメモリモデルを想定していないので，本当はアルゴリズムのバグではなく誤用である．しかし，ここではピーターソンのアルゴリズムを，このようなメモリモデルに対応するように修正することをデバッグと呼ぶことにする．

例である．それでも，ユーザが書いた Promela モデル上の反例としても最短な反例の十分な近似になっている．

次に，本反例可視化ソフトウェアが赤色でハイライトする，バグの原因になり得る WRITE に注目する．プログラム順序では正しく動作するプログラムのモデルを前提としているので，バグの原因は，いずれかの共有変数アクセス命令でプログラム順序とメモリ順序が入れ替わることに絞られる．そこで，順序が入れ替わった実行ステップで，どの命令が入れ替わったかを可視化する，赤色のハイライトを積極的に利用することにする．

### 4.2 ピーターソンによる相互排除アルゴリズム

2 章で述べたように，図 1 のピーターソンの相互排除アルゴリズムでは，プログラム順序とは異なるメモリ順序で実行されると両方のスレッドが同時にクリティカルセクションに入ってしまう．以下では，ピーターソンの相互排除アルゴリズムのモデル（図 2）に対して，本反例可視化ソフトウェアを使ったデバッグの手順を述べる．

(1) SPIN が出力した最短の反例を本反例可視化ソフトウェアにロードして，最初のスレッドがクリティカルセクションに入った実行ステップまで進める．この実行ステップでは，スレッド T1 が図 2 の 25 行目の READ(want0) == 0 の条件成立によってクリティカルセクションに入っているのが分かる．この実行ステップで，初めて赤色でハイライトされる WRITE が現れた（**図 5**）．特に，図 2 の 21 行目のスレッド T1 による WRITE(want1, 1) が赤色でハイライトされており，共有変数上ではまだスレッド T1 がクリティカルセクションに入ることを表明したことになっていないと分かる．これはエラーの原因になり得る．

(2) さらに実行ステップを進めると，今度はスレッド T0 の実行が進み，スレッド T0 が図 2 の 14 行目の，クリティカルセクションのガード文 READ(want1) == 0 を実行する（**図 6**）．このとき，スレッド T0 が実行した WRITE だけでなく，スレッド T1 が実行した，図 2 の 21 行目の WRITE(want1, 1) も赤色でハイライトされた．これは，スレッド T1 のクリティカルセクションに入る表明がスレッド T0 に届かなかったことを表している．実際，共有変数ビューで変数 want1 の値を確認すると 0 のままであった．

(3) ここまでの理解で，図 2 の 21 行目の WRITE(want1, 1) と 25 行目の READ(want0) の順序が入れ替わることがエラーの原因だと分かる．そこで，それを禁止するように，図 2 の 23 行目の if 文の直前に FENCE 命令を挿入する．二つのスレッドは対称なので，スレッド T0 にも同様に FENCE 命令を挿入する．このように修正したモデルでモデル検査すると，エラーが報告され新しい反例が得られた．

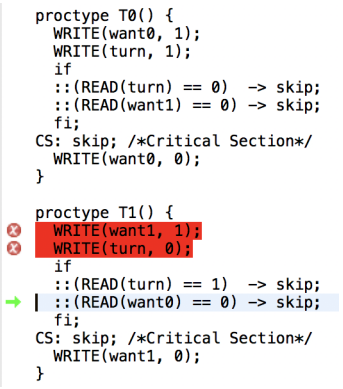



図 5: スレッド T1 がクリティカルセクションに入る実行ステップ

**Fig. 5** Execution step in which thread T1 enters the critical section.

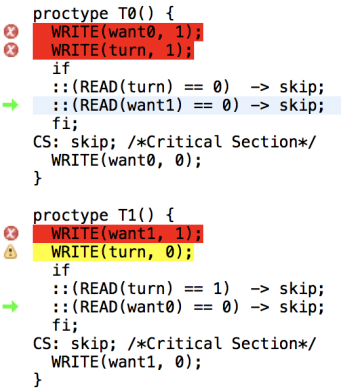



図 6: スレッド T0 がクリティカルセクションに入る実行ステップ

**Fig. 6** Execution step in which thread T0 enters the critical section.

(4) 新しい反例を本反例可視化ソフトウェアにロードして，実行ステップを進めると**図 7** で最初に赤色のマーカーが表示された．この実行ステップで実行が進んだスレッドは T1 だった．したがって，このハイライトは図 2 の 21 行目の WRITE(want1, 1) を追い越して 22 行目の WRITE(turn, 0) が共有変数に反映されたことを示している．このとき，スレッド T0 の実行した図 2 の 11 行目の WRITE(turn, 1) は黄色にハイライトされており，まだ共有変数に反映されていないことが分かる．この WRITE(turn, 1) が共有変数に反映されると，turn は 1 になる．そうすると，スレッド T1 はいつでもクリティカルセクションに入れるようになる．一方，スレッド T1 による図 2 の 21 行目の WRITE(want1, 1) はまだ共有メモリに反映されていない．そのため，スレッド T0 もクリティカルセクションに入れる状態にある．実際，この先の実行でスレッド T0 がクリティカルセクションに入り，続いてスレッド T1 が READ(turn) == 1 の条件成立によりクリティカルセクションに入った．

```
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);
}
```

図 7: 新しい反例で最初に赤色のハイライトが表示される実行ステップ

**Fig. 7** Execution step in which the first red highlight is displayed in the new counterexample.

(5) このエラーの原因は，スレッド T1 が共有変数 want1 への WRITE が共有変数に反映される前に turn に書き込んでしまったことである．そのため，クリティカルセクションに入ることが可能な状態のスレッド T0 によって turn が上書きされてしまい，スレッド T1 もクリティカルセクションに入れるようになってしまった．そこで，WRITE の順序が入れ替わるのを禁止するように，図 2 の 21 行目の WRITE(want1, 1) と 22 行目の WRITE(turn, 0) の間に FENCE 命令を挿入する．スレッド T0 にも同様に FENCE 命令を挿入する．このように修正したモデルでモデル検査すると，エラーが報告されなかった．

### 4.3 並行 GC Staccato

Staccato は，ヒープ領域を FROM 空間と TO 空間の二つの部分空間に分けて管理するアルゴリズムである．コレクタは FROM 空間からアプリケーションのスレッド（ミューテータ）が参照可能なオブジェクトのみを TO 空間にコピーする．その間もミューテータはオブジェクトを使い続けるため，ミューテータが常に最新のオブジェクトにアクセスできるよう，適切な同期が必要になる．

Staccato は，プログラム順序と異なるメモリ順序で実行されることも想定しており，アルゴリズム中にフェンス命令が使われている．しかし，Staccato にはミューテータの書いた値が TO 空間にコピーされないことがあるバグがあることが指摘されている [4]．

本実験では，Staccato のモデルを作ってバグを再現し，本反例可視化ソフトウェアを使ってその原因を理解した．以下では，本反例可視化ソフトウェアをどのように用いたかを述べる．

SPIN が出力した最短の反例を本反例可視化ソフトウェアにロードして，ステップ実行の様子を確認した．この反

```
    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();
```

図 8: Staccato のバグの原因

**Fig. 8** Cause of the bug of Staccato.

例では，実行全体で，モデルにある 17 個の WRITE のうちの 3 個が，合計で 13 回赤くハイライトされた．その中で，**図 8** の 2 行目で赤くハイライトされている WRITE(from_data, val) がバグの原因だった．この WRITE は，ミューテータによる FROM 空間を表す共有変数 from_data への書き込みである．この WRITE の反映が遅れたため，コレクタは新しい値を観測できず，図 8 の 20 行目の WRITE(to_data, READ(from_data)) で，from_data から古い値を読み出して TO 空間にコピーしてしまった．これは本反例可視化ソフトウェアで次のように表示されている．コレクタが，from_data から古い値を読み出したために，同じ変数に対する未反映の WRITE である 2 行目の WRITE が赤くハイライトされている．

### 4.4 議論

ピーターソンの相互排除の実験では，赤色のハイライト表示が有用であった．最初の反例の中で WRITE が赤色にハイライトされた実行ステップは，上記の 2 回だけであり，両方ともエラーに強く関連していた．修正後のモデルの反例では，上記のステップ以降にもう 1 回 WRITE が赤色にハイライトされる実行ステップがあったが，WRITE が赤色にハイライトされる実行ステップは十分に少なく，そのほとんどがエラーに強く関連していたと言える．並行 GC Staccato でも，バグの原因となる WRITE は赤色のハイライトで示されており，また，赤色のハイライトは 13 回と，ユーザが個々に確認できる程度の数であった．

しかし，ピーターソンの相互排除の実験の最初の反例では，同じ実行ステップで変数 turn への WRITE も赤色でハイライトされていた．これは，デバッグの手順 (3) と (5) で修正した両方のエラーが混ざって表示されていたことになる．また，Staccato の実験では，多くのバグに関係ない未反映の WRITE があり，その一部は赤色でハイライトされた．さらに複雑なアルゴリズムのデバッグする場合は，バグに関係ない未反映の WRITE が増えると考えられる．

これは，最短の反例として実行ステップ数のが最小の反

例を使ったからである．WRITE を共有変数に反映させると実行ステップ数が増えるので，結果としてエラーとは関係ない WRITE も共有変数への反映が遅れる反例が選ばれた．MMLib を含む Promela モデル上での最短の反例ではなく，意味的に最も単純な反例を生成することができれば，より簡単にバグを理解できる．これは今後の課題である．

## 5. 既存の反例可視化ソフトウェアとの比較

iSPIN[8] は SPIN 標準の反例可視化ソフトウェアである．MMLib には特化していないが，ユーザが MMLib の内部構造を理解していれば，iSPIN を使っても効率よくデバッグできるかどうか検討する．

iSPIN はデバッグのために，反例の 1 文単位でステップ実行の様子を表示する．ただし，Never Claim を実行する文はステップとして扱わない．**図 9** に，デバッグ用の GUI を示す．GUI は，ステップ実行を制御する上段，モデルとシーケンス図を表示する中段，反例と変数，チャネルを表示する下段に分かれている．図 9 の中段左にはモデルがそのまま表示されており，ユーザはこのモデルと他の部分に表示される情報との対応を取りながらデバッグする．以下では，いくつかの観点で本反例可視化ソフトウェアと iSPIN を比較する．

### 5.1 反例の表示

図 9 の下段中央には反例の実行ステップが表示される．これは，SPIN が出力する反例がほとんどそのまま表示される．例えば，表示されている 7 行目は，図 3 の 1 行目と同じである． **図 ??**の反例の表示は図 3 の反例よりもシンプルなように見えるが，これは図 **??**の反例は SPIN に多くのオプションを与えて生成したからである． この表示は，ステップ実行に連動して範囲が切り替わり，現在の実行ステップの左端のステップ数が黄色でハイライトされる．しかし，本研究の反例可視化ソフトウェアのようにモデル上に実行位置を視覚的に表示する機能はない．

本研究の反例可視化ソフトウェアは，同時には反例中のどれか一つの実行ステップしか表示されない．それに対して，iSPIN で反例がそのまま表示されるので，現在の実行ステップの前後の実行ステップも同時に見ることができる．これは MMLib を使っていないモデルでは便利である．しかし，MMLib を使ったモデルでは，一つの共有変数アクセス命令の実行が多数の実行ステップに展開されるため，共有変数アクセス命令が実行された周辺では前後の実行ステップが表示されてもあまり意味がない．

### 5.2 変数の値の表示

反例中に表示される変数の一覧は下段左に，チャネルの一覧は下段右に表示されている．これらの表示はステップ実行と連動して切り替わる． ユーザは，この情報から各スレッドが観測できる共有変数の値を読み解くことになる．しかし， MMLib は多数の内部的な変数を使っているので，MMLib の内部構造を知っていても，目的の変数を探すのは手間がかかる．MMLib では各スレッドから観測できる変数の値は，複数の内部的な変数の値に依存して決まるので，それを知るにはなおさら手間がかかる．

### 5.3 フィルタリング機能

iSPIN では，図 9 の右上のボタンの左隣のテキストボックスに正規表現を記述することで，表示する情報にフィルタをかけることができる．例えば，一番上の「process ids」というテキストボックスに「2」と記述すると，SPIN がプロセスに割り振る ID の中に「2」を含むプロセスの実行のみが表示される．

この機能を用いて特定のプロセスの動きや，特定の共有変数への WRITE が挿入されるキューに注目してログを見ることができる．しかし，この機能を使っても，反例の中から MMLib の内部状態に関する実行ステップや変数を隠蔽して，MMLib を使う Promela モデルの視点で必要な情報だけ得ることは難しい．

### 5.4 シーケンス図

図 9 の中段左にはシーケンス図が表示される．シーケンス図には，左から順に ID 順に スレッド が実行したチャネルに対する操作が，ステップ実行の時系列に沿って表示される．黄色の四角一つがチャネルに対する操作一つを表す．例えば，一番上の四角はスレッド T1 が 11 番目のチャネルに 1 を挿入したことを表している．赤色の破線は，現在の実行ステップの位置を表している．

MMLib では，チャネルを用いて未反映の WRITE を保持するキューを実装している．そのため，MMLib の実装を知っていれば，シーケンス図から WRITE がキューに挿入されたり取り出されたりするタイミングを読み解くことができる．シーケンス図が，MMLib の内部実装ではなく，プログラムのモデル中に現れる変数名など名前を表示すれば，本反例可視化ソフトウェアと相補的に組合わせて利用できると考えられる．本反例可視化ソフトウェアにそのようなシーケンス図を表示する機能を追加することは今後の課題である．

## 6. 関連研究

メモリモデルを考慮したモデル検査は盛んに行われているが，反例の理解を補助する仕組みは整備されていない．Abe らは，McSPIN というメモリ順序を考慮してモデル検査する手法を開発している [9]．McSPIN は，C 言語のコードを中間言語に変換した上で，メモリ順序を考慮した Promela コードに変換し，SPIN によってモデル検査をする．変換後の Promela コードを SPIN でモデル検査する

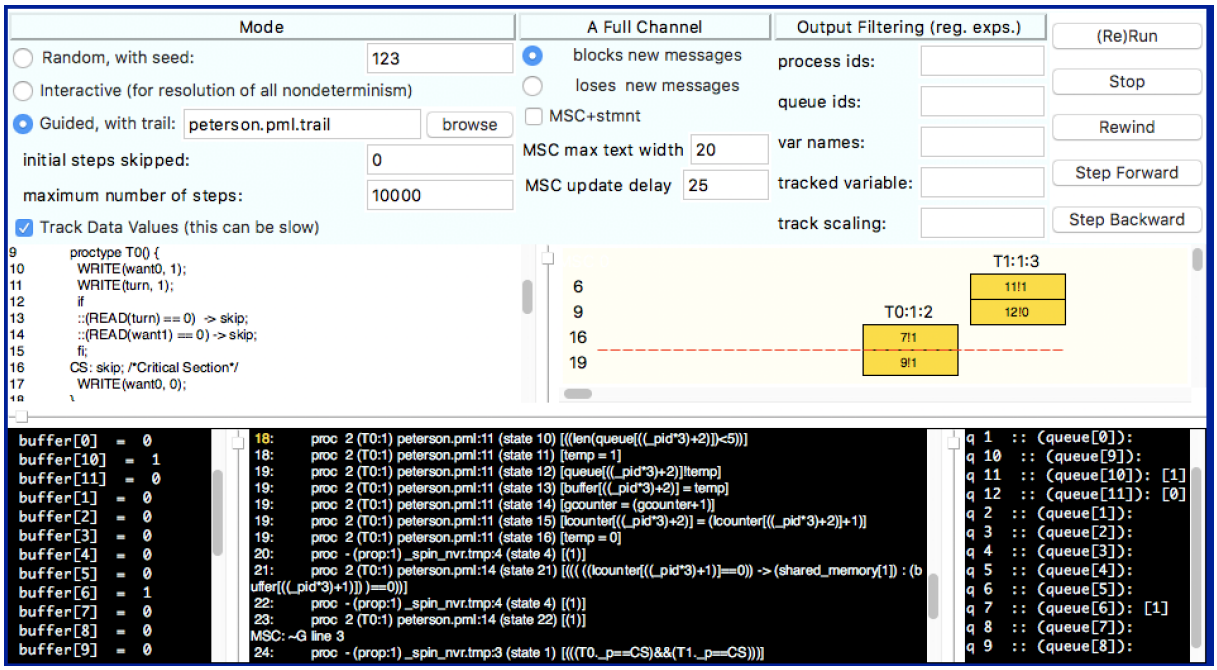



図 9: iSPIN

**Fig. 9** iSPIN

と，反例に相当するモデル検査の実行トレースが出力される．しかし，出力されるのは中間言語の実行トレースであるため，ユーザが作成する C 言語のコードと対応していない．そのため，ユーザが実行トレースを読み解いてデバッグを行うのは困難である．Travkin らも，WEAK2SC というメモリ順序を考慮してモデル検査する手法を開発している [10], [11]．WEAK2SC では，C/C++のコードを LLVM の IR コードを介して，メモリ順序を考慮した Promela モデルに変換しモデル検査する．変換後の Promela コードは，goto 文で実行が制御される．そのため，反例と変換前のモデルとの対応を取ってデバッグを行うのは困難である．これらに対して，本研究の反例可視化ソフトウェアは，ユーザが記述した Promela モデルの上でメモリ順序を表示する．そのため，反例を理解しやすい．

メモリモデルを考慮したものでなければ，モデル検査の反例を可視化する研究は行われている．5 章で比較した iSPIN[8] は，SPIN 標準の反例可視化ソフトウェアである．Pakonen らは，ハードウェアのロジックをモデル検査するためのツール MODCHK と，LTL 中の反例の理解に重要な箇所を可視化するソフトウェアを開発している [12]．MODCHK では，反例のステップ実行をハードウェアのロジックを表すモデル上で可視化する．UPPAAL は，組み込みシステムのモデリングやシミュレーション，モデル検査を行う統合ツール環境である [13]．UPPAAL では，モデル検査の反例をシーケンスグラフやオートマトンのグラフ上で可視化する．Aljazzar らは，モデル検査器 PRISM や MRMC の反例を可視化するツール DiPro を開発している [14]．DiPro では，モデル検査の際に，モデルの状態を網羅したオートマトンのグラフを表示し，その上で実行の様子を可視化する．プログラム順序の実行ではエラーが起こらないという前提で，メモリモデルを考慮したモデル検査を行う場合は，どんな実行パスを通ったかよりも，どのタイミングでどのメモリアクセス命令がメモリに反映されたかの方がデバッグに重要である．そのため，本研究では実際のモデルをそのまま表示し，その上にメモリ順序を可視化するという手法を採用した．

反例に限らず，プログラムの実行の可視化の研究も行われている．Terada らは，プログラムのログから得た関数呼び出しの情報を可視化してデバッグを支援する学生向けデバッグツール ETV を開発している [15]．ETV ではプログラム上に情報を可視化する手法を採用している．丹野らも，プログラム上に情報を可視化するデバッガを開発している [16]．丹野らは，インタラクティブなプログラムのデバッグのために，最近高い頻度で実行された行を視覚的に表示する．本研究でも Promela のモデル上に情報を可視化するが，本研究では，メモリ順序の情報を可視化した．

Czyz らは，プログラムのログからクラスやメソッドの呼び出しなどの関係を表すグラフや，実行のシーケンスグラフを生成してデバッグを支援するデバッグツール JIVE を開発している [17]．これは，大規模なプログラムの動作を俯瞰して理解するためには有用である．本研究が対象とするメモリモデルを考慮したデバッグでは，微視的な視点からモデルの動作を理解する方が重要であるため，本反例可視化ソフトウェアでは Promela のモデル上で反例をステップ実行して表示することにした．

## 7. おわりに

メモリ順序を考慮したモデル検査をするためのライブラリ MMLib の反例可視化ソフトウェアを開発した．本反例可視化ソフトウェアは，反例に沿ってステップ実行する様子をモデル上に可視化する．本反例可視化ソフトウェアを使用することで，ユーザは MMLib の反例を読み解くことなく，共有変数アクセス命令が共有変数に反映された順序などの情報を得ることができる．ケーススタディとして，本反例可視化ソフトウェアを使い，プログラム順序とメモリ順序が異なるようなメモリモデルで，ピーターソンの 相互排除 アルゴリズムで排他制御できるようにするためのアルゴリズムの修正や，並行 GC Staccato のバグの理解を行った． このケースタディから，プログラム順序とは異なる順序で共有変数に反映された共有変数アクセス命令を，それが確定した実行ステップだけでハイライトするのが有用であることが分かった．

効率的なデバッグには単純な反例の生成が欠かせない．これについて，現在の MMLib を用いたデバッグでは，SPIN の最短の反例を出力する機能に頼っている．しかし，SPIN は MMLib を展開したモデル上での最短の反例を生成し，それが必ずしも最も単純な反例とは限らない．特に，共有変数アクセス命令の反映順序は入れ替わる傾向にあり，本反例可視化ソフトウェアでは多くの共有変数アクセス命令がハイライトされるという問題がある．意味的に最も簡単な反例を生成する仕組みを作り，この問題を解決するのが今後の課題である．

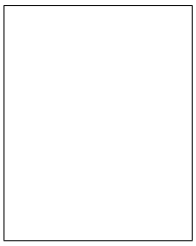
**謝辞** 本研究を行うにあたり，日本電信電話株式会社の丹野治門氏にさまざまなご教示を頂いたことを深く感謝致します．本研究の一部は，JSPS 科研費 16K00103 の助成を受けたものです．

### 参考文献

[1] 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).
[2] 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).
[3] Holzmann, G. J.: *The SPIN Model Checker*, Addison-Wesley (2003).
[4] Ugawa, T., Abe, T. and Maeda, T.: Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models, *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA, pp. 53:1–53:26 (2017).
[5] Iiboshi, H. and Ugawa, T.: Towards Model Checking Library for Persistent Data Structures, *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 119–120 (2018).
[6] Peterson, G. L.: Myths about the mutual exclusion problem, *Information Processing Letters*, Vol. 12, No. 3, pp. 115–116 (1981).
[7] 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).
[8] spinroot.com: Getting Started: Using iSpin, spinroot.com (online), available from ⟨http://spinroot.com/spin/Man/3_SpinGUI.html⟩ (accessed 2018-12-14).
[9] 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).
[10] Wehrheim, H. and Travkin, O.: TSO to SC via Symbolic Execution, *Proc. of HVC*, LNCS, Vol. 9434, pp. 104–119 (2015).
[11] Travkin, O. and Wehrheim, H.: Verification of Concurrent Programs on Weak Memory Models, *Proc. of ICTAC*, LNCS, Vol. 9965, pp. 3–24 (2016).
[12] Pakonen, A., Buzhinsky, I. and Vyatkin, V.: Counterexample Visualization and Explanation for Function Block Diagrams, *2018 IEEE 16th International Conference on Industrial Informatics* (2018).
[13] Behrmann, G., David, A., Guldstrand Larsen, K., Håkansson, J., Pettersson, P., yi, W. and Hendriks, M.: Uppaal 4.0, *Third International Conference on the Quantitative Evaluation of Systems, QEST 2006*, pp. 125–126 (2006).
[14] Aljazzar, H., Leitner-Fischer, F., Leue, S. and Simeonov, D.: DiPro : A Tool for Probabilistic Counterexample Generation, *Model Checking Software*, LNCS, Springer Berlin Heidelberg, pp. 183–187 (2011).
[15] Terada, M.: ETV: A Program Trace Player for Students, *SIGCSE Bull.*, Vol. 37, No. 3, pp. 118–122 (2005).
[16] 丹野治門，岩崎英哉：プログラムを停止させないデバッグを可能とする手法の提案，日本ソフトウェア科学会第 35 回大会講演論文集 (2018).
[17] Czyz, J. K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, ACM, pp. 31–35 (2007).

**松元 稿如** （学生会員）

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．