並行プログラムのPartial Store Orderingでの実行をモデル検査するためのReleaseメモリバリア

鵜川 始陽[1,a)] 松元 稿如[1,b)] 飯干 寛幸[1,c)]

概要：我々は，並行プログラムの弱いメモリ一貫性モデルでの振舞いをモデル検査するためのモデル集（以下，モデル検査ライブラリ）であるMMLibを開発している．SPINモデル検査器で検査するために，Promelaで記述されたモデルに対して，共有変数のリードとライトを，MMLibが提供するマクロの呼出しに置き換えるだけで，TSOやPSOに従った振舞いを検査できるようになる．これまでMMLibにはacquire-releaseフェンスが存在しなかったため，それを用いたプログラムを検査できなかった．本研究では，このうちPSOのreleaseフェンスのモデルを実装した．releaseフェンスの使用に誤りがあることが分かっている並行GCをMMLibを使って検査し，誤りが検出されることを確認した．

キーワード：メモリモデル，モデル検査

1. はじめに

近年のプロセッサでは，ロード命令やストア命令などのメモリアクセス命令の効果がプログラムに書かれている順にメモリに反映されるとは限らない．プログラムに書かれているメモリアクセス命令の順序を**プログラム順序**と呼び，メモリに反映される順序を**メモリ順序**と呼ぶ．プログラム順序とメモリ順序が異なるのは，実行を高速にするために，プロセッサがメモリからの応答を待たずに依存がない命令を先に実行したり，プロセッサコアとメモリの間にあるバッファでメモリアクセスの要求を入れ替えたり結合したりする可能性があるからである．プログラム順序に対して，どのようなメモリ順序が起こる可能性があるかは，プロセッサの**メモリモデル**によって規定される．プログラム順序と異なるメモリ順序の可能性があるメモリモデルを**弱いメモリモデル**と言う．

弱いメモリモデル上のCPUを対象とする並行アルゴリズムは，メモリモデルを考慮して設計する必要がある．あるスレッドの実行は，別のスレッドからはメモリの変化によって観測される．そのため弱いメモリモデルでは別のスレッドからはプログラム順序と異なる順序で実行されたかのように見える．並行アルゴリズムは，メモリモデルによって許されるどのようなメモリ順序でプログラムが実行されたとしても，正しく動作するように設計されている必要がある．

メモリ順序を制御するために，プロセッサはメモリバリア命令，あるいは**フェンス命令**と呼ばれる命令を提供している．フェンス命令は，メモリアクセス命令との間のプログラム順序がメモリ順序で保存される命令である．以降，ある命令間の

[1] 高知工科大学
[a)] ugawa.tomoharu@kochi-tech.ac.jp
[b)] matsumoto@plas.info.kochi-tech.ac.jp
[c)] iiboshi@plas.info.kochi-tech.ac.jp

プログラム順序がメモリ順序で保存されることを，単に命令の**順序が保存される**と言い，命令の順序が保存されないとき，**入れ替わる可能性がある**と言う．フェンス命令によって，フェンス命令を挟んで先行するメモリアクセス命令と，後続のメモリアクセス命令の間の順序が保存される．さらに，プロセッサによっては，フェンス命令に先行する命令と後続の命令がロード命令かストア命令かの合計 4 通りの組合わせのうち，一部の組合わせでのみ命令の順序が保存するような弱いフェンス命令を提供しているものがある [5], [7], [8].

フェンス命令を使えば起こり得るメモリ順序を制限することができるが，パフォーマンスが低下する．アルゴリズムの設計者は，可能な限りフェンス命令の使用を減らし，弱いフェンス命令を使うようにすることで，高性能なアルゴリズムを設計することができる．一方で，フェンス命令を減らし過ぎたり，弱過ぎるフェンス命令を使うと正しくない実行が起こる可能性がある．これはバグである．

並行アルゴリズムのバグを探すには，手続型のモデル記述言語を使った有界モデル検査が有効である．アルゴリズムの疑似コードを手続型のモデル記述言語で記述し，その中にプログラムの不変条件を表す `assert` 文を挿入したり，プログラムの仕様を論理式で記述すると，モデル検査器を使って不変条件や仕様が成り立つかどうかを自動的に検査することができる．

我々は，モデル検査器 SPIN[6] を使ってメモリモデルに関連したバグを探すためのライブラリ MMLib を開発している [10], [16]．SPIN は広く使われているモデル検査器である．SPIN には，Promela という手続型のモデル記述言語で記述されたモデルを与えて検査することができる．しかし，Promela の意味論では共有変数への書込みが即座に他のスレッドから見える状態になるため，そのままではメモリモデルを考慮いたモデル検査ができない．そこで我々は，メモリ順序に影響するハードウェアを Promela でモデル化して，アルゴリズムのモデルと結合できるモデルの部品として提供している．我々はこのモデルの部品を**モデル検査ライブラリ**と呼んでいる．

表 1 acquire フェンスと release フェンス

先行命令	ロード	ロード	ストア	ストア
後続命令	ロード	ストア	ロード	ストア
acquire	保存	保存		
release		保存		保存

SPIN は，Promela で記述されたモデルをオートマトンに変換し，その全状態を網羅的に探索する．そして，その中に不変条件や仕様を満たしていない状態があれば，バグとして報告する．SPIN が作るオートマトンは非常に大きくなり，そのためにモデル検査には大量のメモリを要する．これは，モデル中の各スレッドの実行位置（プログラムカウンタの値）や各変数の値が異なれば，オートマトン上で異なる状態になるためである．そのため，ライブラリの開発にあたっては，同じ内容を表現する内部状態を可能な限り正規化し，ライブラリの内部状態を減らすことが重要になる．

本研究では，MMLib で弱いフェンス命令の一種である acquire-release フェンスを使えるようにする．acquire フェンスと release フェンスは表 1 に示すような効果を持つ．これらのフェンス命令は生産者消費者モデルのように，二つのスレッドが共有変数を用いて排他制御する場合に有用である．例えば ARM プロッサは acquire フェンスとロード命令を組合わせた `LDA` 命令や，release フェンスとストア命令を組合わせた `STL` 命令を提供している [5].

MMLib がサポートするメモリモデルには，プログラム順序とメモリ順序常に一致するシーケンシャルコンシステンシ（SC）の他，x86 プロセッサで採用されているトータルストアオーダリング（TSO）と，さらに弱いパーシャルストアオーダリング（PSO）がある．このうち，acquire-release フェンスと組合わせて意味があるのは，PSO の release フェンスだけである（2 章）.

MMLib は変数毎に FIFO キューを用意することで，PSO で起こり得るメモリ順序を再現している（3 章）．release フェンスの実装では，プログラム順に実行されたストア命令と release フェンスの列を release フェンスで区画に区切り，その区画内

表 2 メモリモデル

先行命令	ロード	ロード	ストア	ストア
後続命令	ロード	ストア	ロード	ストア
SC	保存	保存	保存	保存
TSO	保存	保存		保存
PSO	保存	保存		

でのみストア命令が入れ替わることを許すようにした．そのために，release フェンスが実行されると，全てのキューに区画の区切を示す特別なトークンを挿入した（4 章）．このようにして開発したライブラリを使って，実際に acquire-release フェンスを間違えて使っているアルゴリズムをモデル検査した．その結果，既存の研究で使われたモデル検査器よりも少ないメモリでバグを発見できることを確認した（5 章）．

2. メモリモデル

2.1 メモリ順序

MMLib は，SC，TSO，PSO の 3 つのメモリモデルをサポートしている．表 2 に各メモリモデルで命令の順序が保存される条件を示す．表 2 の空欄になっている先行命令と後続命令の組合わせは，順序が入れ替わってメモリに反映される可能性がある．ただし，どのメモリモデルでも 1 スレッドのプログラムではプログラム順序通りの実行と同じ結果が得られるように，次の例外がある．

- 同じスレッドからの同じ変数に対するストアはプログラム順序が保存される．
- 同じスレッドからの同じ変数に対するストア W と後続のロード R が入れ替わったとしても，R は W で書き込んだ値を読み出す．これを**ストアロードフォワーディング**と呼ぶ．

2.2 acquire-release フェンス

既存の MMLib に実装されているフェンス命令は，フェンス命令を挟んだ全ての命令の組合わせで順序を保存するような，強いフェンス命令だけである．しかし，高い性能を追求するアルゴリズムでは，acquire フェンスや release フェンスのような弱いフェンス命令を用いることがある．そのようなアルゴリズムのバグを探すために，本研究では MMLib で acquire-release フェンスを利用できるようにする．

acquire-release フェンスの仕様である表 1 と，各メモリモデルでプログラム順序が保存される条件を示す表 2 を見比べると，acquire-release フェンスが順序を保存すべき命令の組合わせのほとんどは，メモリモデルによって既に順序が保存されていることが分かる．特に，SC と TSO では，acquire-release フェンスが順序を保存すべき全ての組合わせで，メモリモデルによって順序が保存されている．このようなメモリモデルでは，acquire-release フェンスを単に無視すればよい．また同様に，PSO でも acquire フェンスは無視すればよい．

PSO の release フェンスでは，ストア-ストアの組合わせについて，メモリモデルではプログラム順序が保存されないが，release フェンスで順序を保存する必要がある．このとき，ストア-ロードの組合わせは入れ替わる可能性を再現できるようにすることが重要である．そうしなければ，起こりうるメモリ順序を全て再現したことにならず，バグを見落す可能性がある．

3. MMLib

3.1 MMLib を使った Promela モデル

モデル検査器 SPIN は，Promela という言語で記述されたモデルと，そのモデルが満たすべき不変条件や仕様を受けとり，モデル上の可能な全ての実行経路で，それらが満たされているかどうかを調べる．

例えば図 2 は図 1 に示す生産者消費者モデルの C 言語プログラムを Promela で記述したモデルである．図 1 のプログラムでは関数 `A` と `B` は異なるスレッドによって実行されるとする．また，初期状態ではグローバル変数 `Data` と `Done` は共に 0 であるとする．生産者関数 `A` はデータを作って変数 `Data` に入れてから `Done` を 1 にする．ここではデータは常に 1 であるとしている．消費者関数 `B` は `Done` が 1 になるのを待ってから `Data` を読み出す．このプログラムで関数 `A` と `B` がどのようにインタリーブして実行されたとしても，関数 `B` では 1 が読み出せることを確認したいとする．

```
void A() { /* thread A */
  Data = 1;
  Done = 1;
}
void B() { /* thread B */
  int x;
  while (1)
    if (Done == 1) break;
  x = Data;
  /* assert x == 1 */
}
```

図 1 並行プログラムの例

```
proctype A() {
  Data = 1;
  Done = 1;
}
proctype B() {
  int x;
  do ::(Done == 1) -> break
  od
  x = Data;
  assert(x == 1);
}
```

図 2 図 1 の Promela モデル

```
proctype A() {
  WRITE(Data, 1);
  /* FENCE(); */
  WRITE(Done, 1);
}
proctype B() {
  int x;
  do ::(READ(Done) == 1) -> break
  od
  x = READ(Data);
  assert(x == 1);
}
```

図 3 図 1 の MMLib を使ったモデル

図 2 の Promela モデルでは，proctype キーワードを付けた手続き A と B で，並行に実行される手続きを表現している．手続きの記法は図 1 と近いが，while 文が do に変わっている．do 文は::で始まる分岐を複数持ち，::の直後にある条件が満たされている分岐を非決定的に選択して実行するという動作を繰り返す．条件が満たされている分岐がなければ，満たされるまでブロックする．そのため，この記述で while 文とその中の if 文が表現できる．最後に，assert 文により Data から読み出した値が 1 であるかを調べている．この記述により，SPIN はあらゆるインタリーブのパタンを網羅的に調べて，全ての実行パスで assert 文を検査する．

しかし，Promela の意味論では，変数に書き込んだ値は即座に他の手続きから読み出せる．そのため，弱いメモリモデルで起こり得る実行パスまでは調べることができない．例えば，図 2 のモデルでは，スレッド A による Data への書込みと Done への書込みが入れ替わったメモリ順序で実行された場合，スレッド B では Data から 0 が読み出されることがある．しかし，SPIN はこのような可能性を検査しない．

そこで我々は，弱いメモリモデルで起こり得る実行パスを再現するための Promela で記述したモデルの部品である，モデル検査ライブラリ MMLib を開発している．MMLib は，共有変数への読み書きやフェンス命令のための READ，WRITE，FENCE マクロを提供している．MMLib を用いて図 2 のモデルを書き換えると図 3 のようになる．

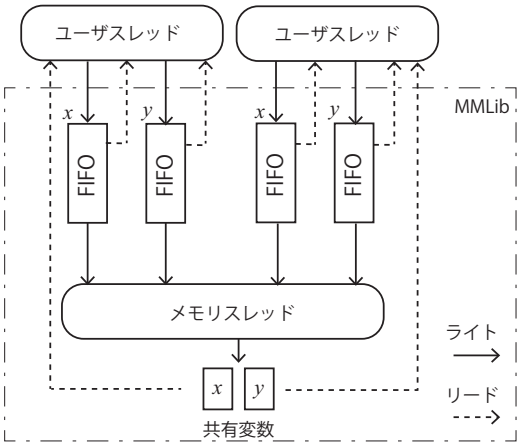

図 4 PSO の実装

例えば，図 3 のモデルと PSO を再現する MMLib を結合してモデル検査すると，スレッド A による二つのストア命令（共有変数への書込みによって起こる）が入れ替わる実行が再現され，スレッド B の assert 文でエラーが検出される．スレッド A でコメントアウトしてある FENCE マクロを有効にすれば，二つのストアはプログラム順序でしかメモリに反映されなくなり，このエラーはなくなる．

3.2 PSO 実装

MMLib では，各スレッドで共有変数毎に FIFO キューを作ることで PSO のメモリ順序を再現している．

図 4 に PSO の実装を示す．MMLib は共有メモリの値を保持するための記憶領域（**共有変数領域**），ユーザスレッドからのストアを遅延させるためのバッファ（**ストアバッファ**），および，ストアバッファに蓄えられた**ストア要求**を共有変数領域に反映させるスレッド（**メモリスレッド**）で構成されている．

WRITE マクロはストア要求をストアバッファに挿入する．ストアバッファは FIFO キューになっており，同じストアバッファに挿入されたストア要求の間では順序が保存される．メモリスレッドは任意のタイミングでいずれかのストアバッファを非決定的に選択して，その先頭のストア要求を共有変数領域に反映させる．そのため，異なるストアバッファに挿入されたストア要求の間では順序が保存されない．ストアバッファは変数毎に用意してあるので，同じスレッドが実行した同じ変数への WRITE の順序は保存されるが，異る変数への WRITE の順序は保存されない．

READ マクロは，バッファを経由せず共有変数領域から値を直接読み出す．これによって，先行する WRITE のストア要求がストアバッファに残っていれば，READ がそれを追い越してメモリに反映されることになる．ただし，ストアロードフォワーディングを実現するために，READ の対象となる変数へのストア要求がストアバッファに存在するときは，ストアバッファ内の値を読出す．

FENCE マクロは，そのスレッドの全てのストアバッファに対して，ストアバッファ内の全てのストア要求を即座にメモリに反映させる．

Promela によるモデルの記述では，共有変数領域は配列で表現されており，ユーザスレッドのモデルでは共有変数は配列のインデックスで識別される．そのため，例えば図 3 のモデルでは，最初に

```
#define Data 0
#define Done 1
```

というような，インデックスを名前に対応させるための定義を書いて使う（図 3 では省略している）．

ストアバッファは，Promela の組込みデータ型であるチャネルを使って実装している．チャネルはスレッド間で通信するための FIFO キューであり，チャネルにメッセージを送受信する操作と，チャネルに入っているメッセージの数を調べる操作がある．チャネルを使えば，配列などで FIFO を実現するよりも効率よくモデル検査できる．

しかし，チャネルは中のメッセージを，取り出さずに見ることはできない．そのため，ストアロードフォワーディングを実現するために，各ストアバッファに，FIFO キューに最近挿入したストア要求の値を記録するための変数も用意している．READ マクロでは，ストアバッファが空でなければ，その変数から読出す．

MMLib のより詳細な実装については文献 [10] を参照されたい．

4. acquire-release フェンスの実装

本研究では MMLib に acquire フェンスを表す AFENCE マクロと release フェンスを表す RFENCE マクロを追加する．2 章で述べたように，PSO の release フェンス以外は acquire-release フェンスは無視してもよいので，PSO 以外のメモリモデルの AFENCE や RFENCE や，PSO でも AFENCE は空のマクロとして実装する．本章の以降では，PSO での RFENCE の実装の詳細を説明する．

PSO のもとでの release フェンスは，release フェンスに先行する WRITE と後続の WRITE の順序を保存しつつ，先行する WRITE と後続の READ の順序が入れ替わることは許すような命令でなければならない．通常の FENCE は，ストアバッファを空にすることで，直ちに FENCE に先行する WRITE を全て完了させる．しかし，これでは FENCE に先行する WRITE と FENCE に続く READ の順序が入れ替わる機会も失ってしまう．そこで，release フェンスでは，それに続くいずれかの WRITE が**メモリに反映される直前**に，それに先行する全ての WRITE をメモリに反映させることにする．

図 5 は，WRITE（W），READ（R），フェンス（F）があるプログラム順序で実行されたときに起こり得るメモリ順序の一つを表している．この図では，矢印が「メモリ」に到達した時に，その命令がメモリに反映されたことを表している．READ は即座にメモリに反映されるので，R からの矢印は必ず水

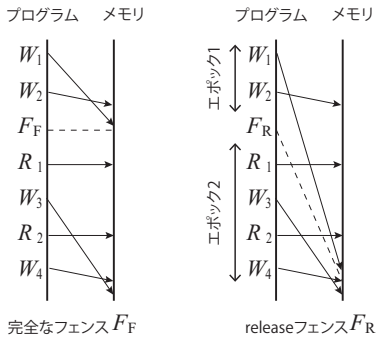

図 5 フェンス命令の動作

平になる．強いフェンス（通常の FENCE）の場合（図の左）は，それに先行する W_1 や W_2 が，フェンス命令 F_F が実行された時にメモリに反映されるため，後続の W_3 や W_4 だけでなく，R_1 や R_2 とも順序が入れ替わる可能性を失ってしまっている．図の右のように，フェンス命令 F_R に続くいずれか先にメモリに反映される WRITE（この場合は W_4）がメモリに反映される直前に，F_R に先行する W_1 や W_2 をメモリに反映すると，R_1 や R_2 とは順序が入れ替わる可能性が残る．

この動作を表現するために，プログラム順序で実行された WRITE 命令と RFENCE（release フェンス）の列を RFENCE の実行で区間に区切る．この区間を**エポック**と呼ぶことにする．図 5 の右では W_1W_2 と W_3W_4 の二つのエポックに区切られる．メモリスレッドは，ストアバッファから新しいエポックのストア要求を取り出すと，そのストア要求を共有変数領域に反映させる前に，全てのストアバッファを調べて古いエポックのストア要求を共有変数領域に反映させる．

エポックの区切を表現するには，エポックの識別子を整数で表現し，各ストア要求にエポックの識別子を持たせることが考えられる．しかし，この方法では RFENCE が実行されるたびに新しいエポック識別子が作られてしまう．検査対象のアルゴリズムにあるイテレーションの回数が決まっていないループは，非決定的にループを抜けるような無限ループでモデル化し，新しい状態が作られなくなるまでループを実行して検査するのが一般的である．その場合，ループの中で RFENCE が使われると，エポック識別子だけが異なる状態が無限に作られて検査が終わらなくなってしまう．

そこで，我々は RFENCE が実行された時に，そのスレッドの全てのストアバッファにエポックの終わりを表すトークン（**RF トークン**と呼ぶ）を挿入することにした．メモリスレッドは，あるストアバッファから RF トークンを取り出すと，他の全てのストアバッファに対して，RF トークが取り出されるまでストア要求を取り出して共有変数領域に反映するという動作を続ける．この処理を**エポック終了処理**と呼ぶことにする．エポック終了処理により，現在のエポックの WRITE で発行されたストア要求を全てメモリに反映することができる．

エポック終了処理は決定的かつ不可分に行う．決定的に行うというのは，つまり，共有変数に付けられたインデックスの順に RF トークンが取り出されるまでストアバッファからストア要求を取り出して反映するということである．これによる利点は，エポック終了処理自体の実行パスが一通りに限られることと，エポック終了処理が他の処理とインタフィーブしなくなることによって，状態数の増加を抑えられることである．

エポック終了処理を決定的に行うと，エポック終了処理でストア要求が反映される順序が固定されてしまい，メモリモデルによって可能な全てのメモリ順序を検査できないと思うかもしれない．しかし，実際にはそのようなことはない．エポック終了処理を始めた時点で，全てのストアバッファの先頭が RF トークンになっているような状態も考えられるからである．このような状態に至る実行を，現在のエポックに対して**最も一般的な実行**と呼ぶことにする．最も一般的な実行には，現在のエポックのストア要求をメモリに反映する全ての順序の実行が含まれる．したがって，メモリモデルによって可能な全てのメモリ順序を検査できる．

別の見方をすると，各エポックについて，最も一般的な実行以外の実行は必要ない．エポック終了処理は，最も一般的な実行以外の実行に対して，最も一般的な実行のいずれかと同じ結果になるように強制する処理である．

5. ケーススタディ

著者らの過去の研究によって，並行コピー GC である Staccato [11] には，acquire フェンスと release フェンスの間違いによるバグがあることが分かっている [14]．本研究では，Staccato のアルゴリズムを本研究で拡張した MMLib を使って Promela でモデル化し，過去の研究と同じバグが発見できるかどうか調べる．また，過去の研究で使ったモデル検査器 McSPIN [3], [4] と検査に要したメモリ量で比較する．

5.1 Staccato

Staccato のアルゴリズムを図 6 に示す．図 6 は，文献 [9], [11] にある疑似コードを著者が整形して，一部，英語による記述しかなかった関数を著者が補ったものである．

Staccato は並行コピー GC なので，コレクタスレッドとミューテータスレッドが並行して実行する．コレクタは `copy` メソッドによりオブジェクト `o` を `allocateToSpace` 関数により割り当てられる領域にコピーする．それと並行して，ミューテータが任意のプログラムを実行する．ただし，ミューテータがオブジェクトからフィールドの値を読み出す時は，`read` 関数を使い，フィールドに書き込む時は `write` 関数を使う．これらの関数の引数 `o` はオブジェクト，`f` はフィールドのオフセット，`v` は書き込む値である．さらに．ミューテータは定期的に `yieldpoint` 関数を呼び出す．これらの関数の動作の説明は省略する（詳細は文献 [14] 参照）．

Staccato が提案された論文によれば，このアルゴリズムは PSO や，さらに弱いメモリモデルでも正しく動作するように設計されている [11]．しかし，`yieldpoint` 関数にバグがあり PSO では正しく動作しない．Staccato では，コレクタが実行する `copy` 関数の中に，`SoftHandshake` 関数を呼出して全てのミューテータが `yieldpoint` を実行するまで待つ処理が含まれている．この処理は，その時点までにミューテータが `write` 関数を使って行ったストア命令が全てメモリに反映される

```
/* executed by collector */
copy(o):
  /* prepare */
  o.forward ← o | COPYING;
  /* sync */
  SoftHandshake();
  AcquireFence();
  /* copy */
  oo ← AllocateToSpace(sizeof(o));
  CopyBytes(o, oo, sizeof(o));
  oo.forward ← oo;
  /* sync */
  ReleaseFence();
  SoftHandshake();
  /* commit */
  CAS(&o.forward, o | COPYING, oo);

SoftHandshake():
  hs_req ← TRUE;
  while (hs_req = TRUE):
    /* do nothing */

/* executed by mutator */
read(o, f):
  return (o.forward & ~COPYING)[f]

write(o, f, v):
  if ((o.forward & COPYING) ≠ 0):
    CAS(o.forward, o | COPYING, o);
  (o.forward & ~COPYING)[f] ← v;

yieldpoint():
  if (hs_req = TRUE):
    AcquireFence();
    hs_req ← FALSE;
```

図 6 Staccato のアルゴリム

のを待つ処理である．しかし，yieldpoint の中で使っているフェンス命令が acquire フェンスであるため，yieldpoint に先行するストア命令と，yeildpoint 中の hs_req への書込みによるストア命令の順序が入れ替わることがある．その結果，PSO で正しく動作しないことがある．この acquire フェンスを release フェンスに変えることで，バグを修正できる．

5.2 MMLib を使ったモデル化

図 6 を MMLib を使って Promela でモデル化したもの（一部のみ掲載）が図 7 である．このモデルでは，オブジェクトは 1 個しか存在せず，ミューテータの持つ変数 root から参照されていると仮定している．ミューテータのスレッド mutator は，非決定的に yieldpoint を実行するか，root が指すオブジェクトから読み出すか，あるいは，root が指すオブジェクトに VALUE1 か VALUE2 のいずれかを書き込む．書き込んだ時には，その値を lst_wr に覚えておき，読み出した時にその値が読み出せるかどうかを確認している．

SoftHandshake と yeildpoint の処理も，対応するインライン関数でモデル化してある．SoftHandshake 関数の 2 行目の等式は，その条件が満たされるまで待つという Promela 独特の記法であり，疑似コードにある while 文に対応している．

5.3 モデル検査

図 7 のモデルを SPIN を使って検査すると，mutator スレッドの assert 文でエラーが報告された．さらに，反例を解析すると，直前の write 関数で実行したストア命令が遅れており，yieldpoint 関数の hs_req への書込みによるストア命令と順序が入れ替わってメモリに反映されていることが分かった．

図 7 の yieldpoint 関数にある AFENCE を release フェンスを表す RFENCE に書き換えてモデル検査すると，エラーは報告されなくなった．このように，本研究で実装した acquire-release フェンスによって，少なくともこのバグは発見できることが分かった．

```
proctype mutator() {
  mtype readval;
  mtype lst_wr = VALUE0;
  do
  ::yieldpoint();
  ::read(root, readval);
    assert(readval == lst_wr);
  ::write(root, VALUE1); lst_wr = VALUE1;
  ::write(root, VALUE2); lst_wr = VALUE2;
  od
}
inline SoftHandshake() {
  WRITE(hs_req, true);
  (READ(hs_req) == false);
}
inline yeildpoint() {
  if
  ::(READ(hs_req) == true) ->
    AFENCE();
    WRITE(hs_req, false);
  ::else -> skip;
  fi
}
```

図 7 Staccato のモデル

表 3 モデル検査に要したメモリ量

N	メモリ使用量（GB）
1	0.5
2	0.7
3	1.3
4	2.7
5	5.7
文献 [14](参考)	6.7

5.4 検査に要したメモリ量

このモデル検査に要したメモリの量を表 3 に示す．ここに示したメモリ量は，修正したモデルを検査するのに要したメモリ量である．修正前のモデルは，バグを発見した時点で検査を終えるので比較する意味がなく，要したメモリ量は調査していない．なお，MMLib はストアバッファに使うチャネルの容量が有限であり，容量をパラメタ（N）にとる．ここで説明したバグは $N = 1$ で発見された．

表 3 を見ると，MMLib は我々が過去の研究で使った McSPIN に比べて優れているように見える．実際，以下で行う議論を考慮しても，PSO の

もとでの検査に限れば MMLib の方が少ないメモリで広い範囲を検査できると結論できる．しかし，比較は単純ではないことを注意しておく．MMLib と McSPIN は汎用性や有界モデル検査で検査する範囲において違いがあるからである．以降では，これらの違いを説明する．

まず，McSPIN は様々なメモリモデルに対応したモデル検査器である．メモリ順序の制約を論理式で記述することで，どんなメモリモデルのもとでの実行も検査することができる．一方で MMLib はメモリモデル毎に専用の実装が必要であり，我々が用意している SC，TSO，PSO のもとでの実行しか検査できない．

次に，MMLib と McSPIN では有界モデル検査の範囲を決めるパラメタが異なる．MMLib ではストアバッファの容量をパラメタで指定する．ストアバッファが一杯のときに `WRITE` マクロを実行すると，メモリスレッドがストア要求をストアバッファから取り出すまでそのスレッドがブロックする．そのため，ストアバッファの容量を N とすると，プログラム順で N を越える数の同じ変数への `WRITE` を飛び越えるような順序の入替えが起きるような実行は検査できない．一方で McSPIN はループのイテレーション回数の上限を指定し，その上限を越えるような実行は検査できない．我々の過去の研究では，ループのイテレーション回数の上限を 1 回にしている．

最後に，本研究のモデルでは，ミューテータが `read` 関数や `write` 関数を任意の順序で任意の回数呼出して，新しい状態が作られなくなるまで検査を続ける．一方，我々の過去の研究では，ミューテータが `read` 関数や `write` 関数を呼出す順序を固定して検査していた．このようにしたのは，ループのイテレーション回数に上限があったためである．本章で取り扱ったバグは，`write` の後に `read` を実行するだけで発見できるバグであったため，表 3 に載せたメモリの使用量はそのような実行によるものである．

6. 関連研究

Wehrheim らは，MMLib と同様の方法で TSO や PSO のメモリモデルを再現する Promela のモデルを開発している[*1]．また，C/C++のコードを LLVM の IR コードを介して，TSO や PSO のメモリモデルでの動作を再現する Promela モデルに変換する手法も提案している [13], [15]．Tomasco ら [12] や Abdulla ら [1], [2] もメモリモデルを考慮したモデル検査器やモデル検査の仕組みを提案している．しかし，いずれも release フェンスには対応してない．

安部ら [3], [4] は McSPIN というメモリモデルを論理式で記述することで任意のメモリモデルのもとでの動作を検査できるモデル検査器を開発している．release フェンスもメモリモデルの一部として論理式で記述できるが，5 章で示したように，任意のメモリモデルが扱えるという柔軟性と引き換えにモデル検査に本研究よりも多くのメモリを要する．

7. まとめ

本研究では，MMLib に acquire-release フェンスを追加した．特に PSO の release フェンスの実装は自明ではなかった．本研究で拡張した MMLib を用いて，既存研究によって発見されていた，acquire-release フェンスを間違えて使っているバグを発見することができた．

謝辞 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] 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, *19th Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '14)*, IEEE, pp. 332–341 (2014).

[*1] https://github.com/oleg82upb/lina4wm-examples

[4] Abe, T., Ugawa, T., Maeda, T. and Matsumoto, K.: Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models, *Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA '16)*, LNCS, Vol. 9984, pp. 118–135 (2016).

[5] ARM: *ARM Architechture Reference Manual ARMv8, for ARMv8-A architechture profile* (2016).

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

[7] IBM: *Power ISA(TM) Version 3.0* (2015).

[8] Intel: *Intel® 64 and IA-32 Architectures Software Developer's Manual* (2016).

[9] Jones, R., Hosking, A. and Moss, E.: *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Applied Algorithms and Data Structures, Chapman & Hall (2012).

[10] 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).

[11] McCloskey, B., Bacon, D. F., Cheng, P. and Grove, D.: Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors, IBM Research Report RC24505, IBM Research (2008).

[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 FM-CAD* (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] Ugawa, T., Abe, T. and Maeda, T.: Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models, *Proceedings of the ACM on Programming Languages*, Vol. 1, No. OOPSLA, pp. 53:1–53:26 (2017).

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

[16] 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).