B-010

LMNtal 処理系を用いた操作的メモリモデルのシミュレーションと検証
Simulating and Verifying Operational Memory Models using LMNtal

岡 慶樹[1)]　上田 和紀[2)]
Yoshiki Oka　Kazunori Ueda

1　はじめに

共有メモリをもつ並列システムにおいて，メモリアクセスのアーキテクチャ的振る舞いを表すモデルは、メモリモデルと呼ばれる．現代の多くのプロセッサは逐次一貫性 [6] モデルよりも弱いメモリモデルをもち、実際のプロセッサのメモリモデルの形式化やそのもとでのプログラムの振る舞いに関して、多くの研究が行われている．

操作的メモリモデルは、メモリモデルを抽象機械の状態遷移として形式化したものであり，メモリモデルの形式化手法の主な手法の一つである．また、メモリモデルの振る舞いを理解するために、リトマステストと呼ばれる小さな並行プログラム断片がよく用いられる．これは，特定のリトマステストについての可能な実行結果がメモリモデルによって決まるからであり，特に操作的メモリモデルにおいては，その結果を生じる抽象機械のトレースに関する議論が頻繁に生じる．よって，特定の操作的モデルのもとでリトマステストの実行をシミュレートする環境はメモリモデルを理解する上で有用である．しかし、メモリモデルのモデルの追加や変更が簡単に行えるような操作的メモリモデルのシミュレーション環境は非常に少なく，さらにモデルの抽象機械の状態遷移を可視化する環境は著者の知る限り存在しない．

本研究では、汎用モデリング言語 LMNtal によって x86 アーキテクチャの x86-TSO [2]，および Arm アーキテクチャの Flat [3] の二つの操作的メモリモデルを実行可能な形で表現したうえで，いくつかの例題について統合開発環境 LaViT を用いた状態空間の解析を行うことで、操作的メモリモデルを表現し，理解するための環境として LMNtal が有用であることを示す．

2　LMNtal

LMNtal は階層グラフ書き換えに基づく計算モデルであると同時に，汎用的なプログラミング言語・モデリング言語でもある．モデリング言語としての LMNtal は階層グラフ書き換え系を記述する言語であると見ることができるため，複雑な接続関係と階層構造で表される状態，さらにこれら状態間の遷移規則を簡潔に表現することができる．

LMNtal の基本要素であるプロセス P の構文を次に示す．

$$P ::= \mathbf{0} \mid p(X_1,\dots,X_n) \mid P,P \mid \{P\} \mid \texttt{name@@}\ T \text{:-} T$$

p, $X_i (i = 1,\dots,n)$ はそれぞれアトム名，リンク名とよばれる文字列である．P, P はプロセスの並列合成，$\{P\}$ は膜と呼ばれる階層構造に囲まれたプロセスを表す．T :- T はプロセスの書き換え規則を表す．LMNtal のプロセスは階層化されたポートグラフと見ることができる．以降，ルールをもたない LMNtal プロセスを LMNtal グラフとも呼ぶ．LMNtal の厳密な構文，意味論については文献 [1] に委ねる．

2.1　処理系

LMNtal の処理系はコンパイラ (LMNtal コンパイラ) とランタイム (SLIM) からなる．LMNtal コンパイラは LMNtal プログラムを中間コードに変換し，SLIM が中間コードの実行を行う．SLIM にはいくつかの実行モードがある．

通常実行　適用可能なルールを 1 つ選び，書き換えを行うことを繰り返す

非決定的実行　適用可能なルールすべてについて書き換えを行うことを繰り返し，到達可能なプロセスをすべて求める（状態空間の構築）

LTL モデル検査　状態空間が LTL 式を満たすか検査し，満たさなければ反例を出力する

2.2　LaViT

LaViT は LMNtal の統合開発環境である．記述した LMNtal コードについて，SLIM による通常実行，非決定的実行，LTL モデル検査のいずれも行うことができる．特に，非決定的実行および LTL モデル検査については，書き換えによって生成されうるすべてのプロセスを有向グラフとして表示すること（状態空間表示）ができ，さらにグラフの整形や適用されたルールの検索，特定のノードから到達可能なノードの強調など様々な機能が利用できる．

次の LMNtal プログラムから生成される状態空間を LaVit を用いて表示したものを図 1 の左に示す．

```
a.
r1@@ a :- b.
r2@@ b :- c. r3@@ b :- d.
r4@@ d :- y.
r5@@ c :- b. r6@@ c :- x.
r7@@ x :- d.
```

LaViT の状態空間表示では，Transition Abstraction と呼ばれる機能を利用できる．ルール名の集まり S があるとき，S に関する Transition Abstraction は，$p1$ がある S に属するルールによって $p2$ に書き換えられうるとき，$p1$ と $p2$ に相当するノードを一つに（グラフの意味で）縮約する．

Transition Abstraction を使うと，状態空間の構造を保ちつつ興味のある状態遷移だけを表示することができるので，特に状態遷移系を理解するうえで有用である．図 1 の右は，左の状態遷移図について，$S = \{\texttt{r1}, \texttt{r3}, \texttt{r6}\}$ に関する Transition Abstraction を適用した結果である．

1)　京都大学
2)　早稲田大学

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

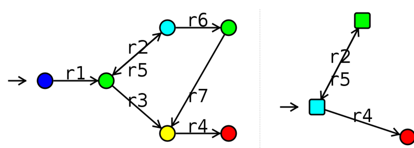

図 1　Transition Abstraction 適用前と後の状態遷移図

SB　　　　　　x86

Thread 0	Thread 1
x = 1 r1 = y	y = 1 r1 = x
Allowed: 0:r1=0∧1:r1=0	

図 2　リトマステスト：SB

3　メモリモデル

マルチコア・システムにおいては，同期されていない複数のプロセッサコアが共有するメモリに対して書き込み・読み込みを行う状況が生じ得る．メモリモデルは共有メモリのアクセスに関するアーキテクチャ的な仕様であり，このような状況において可能な実行結果を規定する．

3.1　リトマステスト

リトマステストはプロセッサのメモリに関する振る舞い（メモリモデル）を理解するために作られる，小さな並行プログラムと，期待される実行結果に関する制約である．図 2 に SB と呼ばれるリトマステストを示す．

`x`, `y`, ... はメモリ位置，`r1`, `r2`, ... はレジスタを表す．また，特に断らない限り，メモリ，レジスタともに初期値はすべて 0 であるとする．

図の右上と表の最下段は，x86 アーキテクチャにおいてスレッド 0 のレジスタ r1 とスレッド 1 のレジスタ r1 がどちらも値 0 を持つような最終結果が生じ得ることを示している．SB はストアバッファによって引き起こされる振る舞いを特徴づけるリトマステストとして知られる [4].

3.2　実行候補

リトマステストの実行により，ある結果（すなわち，各メモリ位置における値および各レジスタの値）が得られたとすると，この結果を実現するような命令間のデータフローがあると考えることができる．

そこで，このようなデータフローをメモリアクセス要求の間の関係として抽象的に表現することが行われる．これは実行候補 (candidate execution) [4] と呼ばれ，実行図と呼ばれる有向グラフによって表現される．

図 2 のリトマステストに対応する実行候補を実行図により表したものを図 3 に示す．

ある実行候補が実際に可能な実行であるかは，メモリモデルによって決まる．実行候補や実行図に関する詳しい説明は文献 [4][5] に委ねる．以降，リトマステストを示す場合はアセンブリ命令や擬似コードの代わりに，実行図を示すこととする．

3.3　x86-TSO

x86-TSO [2] は Owens らによって導入された x86 アーキテクチャのメモリモデルであり，ストアバッファの操作を定義し，アーキテクチャの振る舞いを厳密に記述する．図 4 に x86-TSO の抽象機械を図示したものを示す．

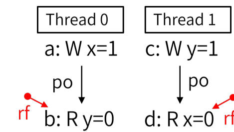

図 3　SB の実行候補

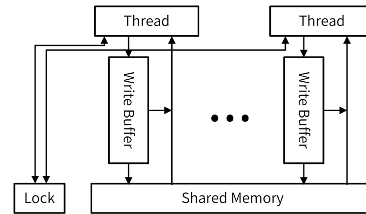

図 4　x86-TSO の抽象機械

x86-TSO の抽象機械は主に共有メモリと n 個のスレッド，さらにその間のストアバッファ（ライトバッファとも呼ばれる）からなる．各スレッドから発行された共有メモリへの書き込み命令は必ず一度ストアバッファに書き込まれるため，後から発行された読み込み命令が書き込み命令を追い越すことがある．詳しくは文献 [2] を参照されたい．

3.4　Flat

Flat [3] は Schur らによって導入された Armv8-A アーキテクチャのメモリモデルである．Flat において，モデルの状態は共有メモリと n 個のスレッドからなる．共有メモリは，各メモリ位置への最新の書き込みを保持する．各スレッドは主に命令インスタンスと呼ばれる命令の実行状態をノードに持つ木からなる．各命令は基本的に独立に進行するが，命令間にデータ依存やアドレス依存がある場合，それらの命令間の実行順序は制約を受ける．命令インスタンスの進行やそのための事前条件は Flat の状態遷移規則によって規定される．詳しくは文献 [3] を参照されたい．

4　メモリモデルの表現手法

本研究では，操作的メモリモデルの抽象機械を LMNtal グラフ，状態遷移規則を LMNtal ルールとして表現した．

4.1　x86-TSO

ストアバッファに対して，次の操作が行えることが必要である．

- 与えられた書き込みの末尾への挿入
- 先頭にある書き込みの取り出し
- 位置 x が与えられたとき，先頭に最も近いような，書き込み先が x であるような書き込みの特定

LMNtal では適切なアトムの接続によりストアバッファを表現することで，ストアバッファに関して必要な操作をそれぞれ一つのルールとして表現することができる．

例を用いて表現方法を説明する．図 7 に示す状態にあるストアバッファは，LMNtal グラフとして次のように表すことができる．

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

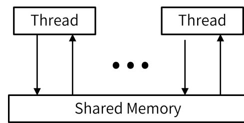

図 5　Flat モデル

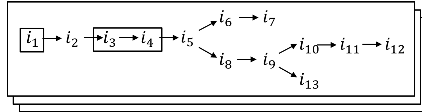
図 6　Flat のスレッドモデル

```
{ c(y,L2,s(c,L3,L4,L2)),
  c(z,L5,s(a,L6,b(L7),L5)),
  c(x,L8,s(b,L4,L6,s(d,L7,L3,L8))). }
```

アトム s はストアバッファに格納される要素を表す．また，アトム b はストアバッファの先頭要素と末尾要素につながるリンクを，アトム c はストアバッファの内容を各位置に制限した場合の，先頭要素と末尾要素につながるリンクを保持する．

ストアバッファに対する操作の一例として，ストアバッファからの読み取りに対応する LMNtal ルールを次に示す．

```
rb @@
not_blocked{ k(K), $t },
{ k(K), t = [r($x1,$a)|T], c($x2,C1,C3),
  s($b,B1,B2,C1,C2), $p[T,C3,B1,B2,C2] }
 :- $x1 == $x2, unary($a), unary($b)
  | not_blocked{ k(K), $t },
    { k(K), t = T, c($x2,C1,C3),
      s($b,B1,B2,C1,C2), $p[T,C3,B1,B2,C2] },
      rf($b, $a).
```

アトム c を使い，位置 x1 に対する最近の書き込みを取り出している．

4.2　Flat

リトマステスト PPOCA（図 11）のスレッド 1 の初期状態における Flat の抽象機械の LMNtal による表現を次に示す．

```
T2={
 regw{val,fd,+CA1,+CA2,+CA3}.
 regw{val,fd,+CD1}.
 po(term,L1)={tail.}.
 po(L1,L2)={mem(r),addr(CA1,y),name(c),
            '*'(R1),j{}},regw{-R1,+D}.
 po(L2,L3)={condbr(D),j{}}.
 po(L3,L4)={mem(w),addr(CA2,z),data(CD1,val),
            name(d),j{}}.
 po(L4,L5)={mem(r),addr(CA3,z),name(e),
            '*'(R2),j{}},regw{-R2,+A4}.
 po(L5,L6)={mem(r),addr(A4,x),name(f),
            '*'(R3),j{}},regw{-R3}.
 po(L6,term)={head.}.
}.
```

命令 e と f の間のアドレス依存関係 (addr) がリンク A4 によって表されていることに注目されたい．

Flat の遷移規則を LMNtal ルールによって表現するには工夫が必要である．Flat のメモリモデルは [3] において逐語的に表現されており，事前条件を充足しているかを判断するアルゴリズムは示されていない．本研究では

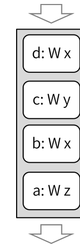

図 7　ストアバッファの状態例

擬似コードによりアルゴリズムを記述したのち，これをもとに LMNtal のルールを作成した．

5　例題

作成したモデルを用いて，並行プログラムの LTL モデル検査やリトマステストの実行過程の検証を行った．なお，リトマステストについてはここに示したもの以外にも 10 種類以上のリトマステストを作成したモデル上で実行し，公理的モデルシミュレータ herd7 [5] による結果と一致することを確かめた．

5.1　SB

x86-TSO におけるリトマステスト SB (図 2) について，LaViT を用いて状態空間を描画したものを図 8 に示す．赤い丸が最終状態，すなわちこれ以上ルール適用を行うことができない状態を表す．それぞれの最終状態は，それぞれ図 9 左に示したような LMNtal グラフである．

図 9 右に，herd7 による SB の実行結果を示す．リトマステスト SB について，SLIM と herd7 が同等の実行結果を出力していることがわかる．

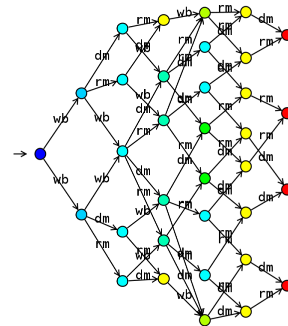
図 8　SB の状態空間

5.2　Dekker のアルゴリズム

作成した x86-TSO のモデルを用いて，x86 における Dekker の排他制御アルゴリズムの性質の検証を行った．メモリフェンスを用いない場合，x86 では Dekker のアルゴリズムは正しく動作しないことが知られている．

次の LTL 式は，公平なスケジューリングのもと，クリティカルセクションに 2 つのスレッドが同時に侵入することがないことを表す．

```
!<>(crit0 && crit1) &&
([](
  (!be0 -> <>pi0) && (!be1 -> <>pi1)
  && <>pv0 && <>pv1
  ) -> [](<>chk0 && <>chk1))
```


Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

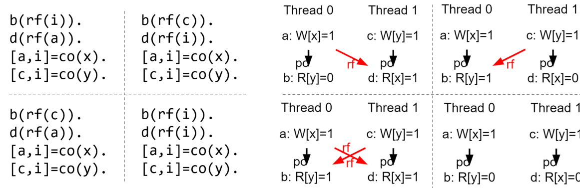

図 9　SB の実行結果（左：SLIM，右：herd7）

SLIM の LTL モデル検査機能を使うことで，次が確かめられた：x86-TSO において，Dekker のアルゴリズムを単純に実装するとこの性質は成立しないが，適切な箇所にメモリバリア命令 `mfence` を挿入すれば成立する．

5.3　CoWR0

CoWR0 は書き込みに続けて読み込みを行う簡単なリトマステストである [4]．Flat では投機的実行による命令のリスタートが生じることが，Transition Abstraction 後の状態遷移図（図 10）からもわかる．

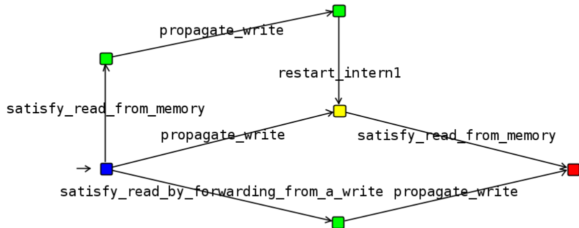

図 10　CoWR0 の状態空間 (Transition Abstraction 後)

5.4　PPOCA

PPOCA は Arm において，フォワーディングに起因する振る舞いを切り出したリトマステストである [4]．SLIM の並列実行機能を用いることで，実行結果を網羅的に求めることができる（図 12 左，4608 状態）．図 12 の右はメモリアクセスとフォワーディングに関するもの以外のすべてのルールについて Transition Abstraction を行った後の状態遷移図である．リトマステストにおいて興味のある実行結果 (これは上から 3 番目の最終状態に対応する) を生じる状態遷移は実質的に一通りであることがわかる．さらに，状態空間表示において各状態を確認すると，結果へ至る遷移は図 11 における $e \to f \to a \to b \to c \to d$ の実行に対応することがわかる．

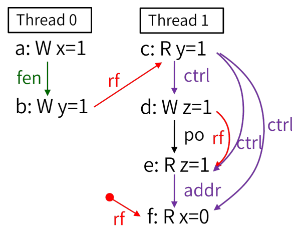

図 11　PPOCA

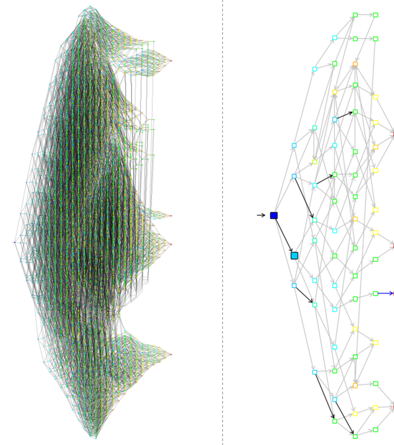

図 12　PPOCA の状態空間

6　まとめと今後の課題

本研究では、汎用モデリング言語 LMNtal によって x86-TSO および Flat の二つの操作的メモリモデルを実行可能な形で表現したうえで，いくつかの例題について統合開発環境 LaViT を用いた状態空間の解析を行うことで、操作的メモリモデルを表現し，理解するための環境として LMNtal および LaViT が有用であることを示した．

現状，SLIM では対話的に適用ルールを選択する（インタラクティブ実行）機能は実装されていない．このような機能を実装することは今後の課題の一つである．さらに，本研究では，Transition Abstraction の代わりに，SLIM の zerostep 機能を用いた部分がある．しかし現状，本機能は試験的であり，明確な仕様が定められていない．本機能を安定させることは，状態数の削減に大きく貢献するため，今後の重要な課題である．

本研究の一部は，科学研究費補助金 23K11057 の援助を得て実施した．

参考文献

[1] Ueda, K.: LMNtal as a hierarchical logic programming language. Theoretical Computer Science, Vol. 410, pp. 4784–4800, 2009.

[2] Owens, S., Sarkar, S. and Sewell, P.: A Better x86 Memory Model: x86-TSO. TPHOLs 2009, LNCS 5674, Springer, pp. 391–407, 2009.

[3] Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S. and Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proceedings of the ACM on Programming Languages, Vol. 2, No.POPL, Article 19, 2018.

[4] Maranget, L., Sarkar, S. and Sewell, P.: A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, 2012.

[5] Alglave, J., Maranget, L. and Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Transactions on Programming Languages and Systems, Vol. 36, No.2, Article 7, 2014.

[6] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, Vol. C-28, pp. 690–691, 1979.

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.