行列計算に基づくモデル検査技術

熊澤 努
株式会社 SRA
kumazawa@sra.co.jp

小田朋宏
株式会社 SRA
tomohiro@sra.co.jp

要旨

モデル検査は，状態遷移系として与えられたソフトウェアのモデルが，望ましい性質を満たすことを形式的に検証する技術である．状態遷移系は，ソフトウェアシステムの振る舞いの記述に適しており，有向グラフで表現することが多い．有向グラフは隣接行列を使って表すことができるので，状態遷移系もまた行列で記述することが可能である．行列は，機械学習などの様々な分野での標準的なツールとして広く利用されている．実用上，*GPU* や分散環境上で大規模な行列の計算が扱われることも多く，演算のための様々なライブラリが整備されている．本論文では，行列計算を使用したモデル検査技法を提案する．提案する検査技法は，並行システムの検証を対象として，モデル検査の主要なアルゴリズムを行列計算で構成する．今後 *GPU* を使った高速化，高性能化が期待できる．本手法の実現可能性を評価するために，試作プログラムを作成してデッドロック検出問題に適用し，実行性能については改善が必要ではあるものの，全ての事例についてデッドロックが検出できることを確認した．

1. はじめに

モデル検査 [1, 2] は形式的検証技術の一つである．ソフトウェアやハードウェアで構成されたシステムの振る舞いや，通信プロトコルの正しさを自動検証するために，モデル検査が広く利用されている．ユーザが与えるモデル検査への入力は，システムの挙動を記述したモデルと，システムが満たすべき性質である．モデルは，クリプキ構造や有限オートマトンなど，有向グラフで表現可能な状態遷移系で記述されることが多い [2, 3]．一方，性質は時相論理式 [2, 3] などの論理式で表現される．モデル検査は，モデルが性質を満たすかどうかを網羅的に調べ，満たさない場合には反例と呼ばれる診断情報を返す．モデル検査には様々な技法が研究されているが，オートマトン理論を用いたモデル検査技法 [4, 5, 6] は，時相論理式を有限オートマトンに変換することで，検証を有向グラフの探索問題に帰着させる．

モデル検査は，並行システムの検証で有効であることが知られている [5, 6]．並行システムは，複数の自律動作するプロセスが相互に通信しながら計算を進める複合的なシステムである．その振る舞いは，各プロセスの挙動の組合せにより構成され，しばしば大規模かつ複雑な有向グラフで記述される．そのため，不具合の検出が難しいことが知られている．本論文では，並行システムの検証を対象として，オートマトン理論を用いたモデル検査技法に注目する．モデル検査の新しい技法として，行列計算を使用した検査アルゴリズムを提案する．

行列は，機械学習や科学技術シミュレーション，ソーシャルネットワークの解析など様々な分野で使われている [7]．例えば，ソーシャルネットワーク解析では，ユーザ間の関係をグラフ構造で表すことが多い．一般にグラフは隣接行列で表すことができるので，ソーシャルネットワークの解析の際に隣接行列が利用される [8]．実際の応用では，しばしば数千万次から数億次に至る規模の行列が扱われる．近年は，GPU を搭載した高性能計算機が手に入れやすくなったことに加えて，高速な行列計算ライブラリを気軽に利用できるようになったことで，大規模な行列の計算を計算機で実行することが可能である．

オートマトン理論を用いたモデル検査は，モデルや性質を有向グラフで表現しその探索問題として解くことができる．本論文では，有向グラフを隣接行列で表し，モデル検査アルゴリズムを隣接行列の演算によって構成す

る．各プロセスの振る舞いを結合したグラフを構成する同期積演算を，行列のクロネッカー積により実現する．探索と反例の生成には，代数的グラフアルゴリズム [9] を使用する．さらに，モデル検査の一手法として，デッドロックの検出も行列演算で構成する．

提案手法の実現可能性を確認するために，行列計算を用いた検証プログラムを試作し，評価実験を行った．実験では，複数のプロセスから構成される並行システムのデッドロック検出問題を対象として，いくつかの異なる規模について，提案手法を用いてデッドロックの検出を行った．その結果，実行速度，メモリ消費の点では改善が必要ではあるものの，全ての問題でデッドロックを検出することができた．

本手法の利点として，モデル検査アルゴリズムを代数的に記述することができることが挙げられる．加えて，行列計算は，GPU 上での実行や分散環境での並列化の実現が期待できる．従来は，モデル検査に特化した並列化や，特定のプロセッサ向けのアルゴリズムが研究されることが多かったが，数値計算や高性能計算分野での研究成果を取り入れた高性能化も可能であると思われる．

本論文の構成は次の通りである．2 節では，関連する先行研究について議論する．3 節で，技術的な背景として，モデル検査の概要を簡単な例を使いながら説明する．4 節で，提案する行列計算を用いたモデル検査手法の詳細を述べる．加えて，同じ技術により実現可能なデッドロック検出技法についても説明する．5 節では，提案手法を実装した試作プログラムの概要を述べ，試作プログラムを用いて行った実験の結果を議論する．6 節で結論と今後の課題を述べる．

2. 関連研究

従来のモデル検査は，記号的モデル検査とオートマトン理論を用いたモデル検査に大きく分類される．記号的モデル検査 [2] は，ソフトウェアの挙動を，順序付き二分決定グラフに代表されるように論理式で記述して検証を行う技法である．一方，オートマトン理論を用いたモデル検査 [4, 10, 2] は，ソフトウェアの挙動を有限オートマトンで明示的に表現し，その受理可能性を判定する．有限オートマトンは有向グラフとみなせるので，効率的なグラフ探索アルゴリズムを使用できる点に特徴がある．加えて，On-the-fly 検査法や半順序簡約法などの，検査の効率を向上させる最適化技術を組込むことで，検査系の実用化に成功している [2, 3]．Spin [5] や LTSA [6] が代表的なモデル検査系である．GPU 上で高速に検査をするための，オートマトン理論を用いた検査アルゴリズム [11] や，On-the-fly 検査法 [12]，半順序簡約法 [13] などの技術も提案されている．オートマトン理論を用いた既存のモデル検査は，ソフトウェアの挙動を有限オートマトンとしてモデル化し，有向グラフ上を網羅的に探索することで，ソフトウェアの挙動に関する性質を検証する技術である．ソフトウェアの挙動を有限オートマトンでモデル化する点が本論文で提案する手法と共通しているが，既存技術が有向グラフ上の探索アルゴリズムを利用するのに対して，提案手法は，隣接行列で表現して，行列演算を用いて検証する点が異なる．

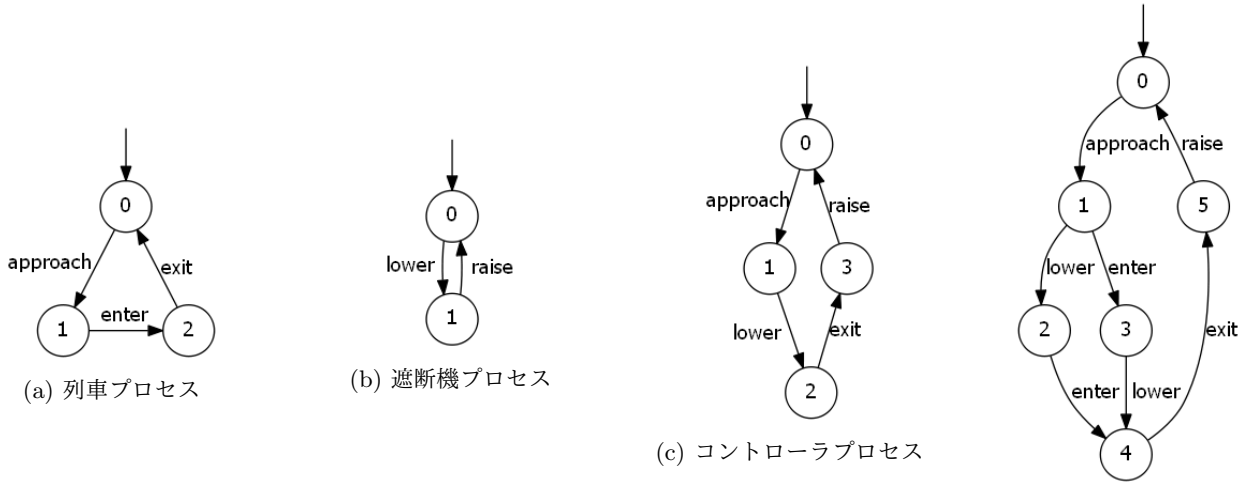
ソフトウェアの検証問題を既存のソルバーを使って解くことで，高速化，効率化を図るアプローチも積極的に研究されている．Alloy [14] や有界モデル検査 [15] は，検証問題を充足可能性判定問題に帰着させ，高速な SAT ソルバーを活用して検証を行う．

確率的モデル検査 [3] は，マルコフ連鎖やマルコフ決定過程を対象とした検証技術である．これらは推移確率行列や状態遷移系で表現できることが知られており，行列計算によりシステムの確率的な挙動の検証が可能である．Wijs 等は，確率的モデル検査を GPU 上で実行する際の効率的な疎行列とベクトルの演算技法を提案した [16]．文献 [17] では，複数のサブシステムから成る複合システムの確率的モデル検査法が提案されている．

3. オートマトン理論を用いたモデル検査

本節では，2 節で紹介したオートマトン理論を用いたモデル検査のうち，Giannakopoulou と Magee による技法 [10] の概要を，簡単な例を使って紹介する．

本論文では，ラベル付き遷移系（LTS） [6] を用いた状態遷移系によって，システムの振る舞いを記述する．LTS は，状態の集合，状態間の遷移関係，初期状態で構成される．遷移関係には，アクションあるいはイベントと呼ばれるラベルが付与されている．初期状態から状態遷移を繰り返すことで，システムが起こすイベントの列を表現できる．図 1 に踏切の遮断機制御システム [3] を LTS で記述したモデルを示す．遮断機制御システムは列車，遮断機，コントローラの 3 つのプロセスから成る．列車プロセス（図 1a）は，踏切付近の列車の動きをモデル化したプロセスである．図 1a で，丸が状態，矢印



(a) 列車プロセス　(b) 遮断機プロセス　(c) コントローラプロセス　(d) システム全体の挙動

図 1. 踏切の遮断機制御システム

が遷移関係，approach などのラベルがイベントである．初期状態は，遷移元のない矢印が指す状態 0 である．このモデルは，列車が踏切に近づき (approach)，次に進入して (enter)，退場する (exit) という一連の挙動をイベントで表現している．遮断機プロセス（図 1b）は，バーを下ろす (lower)，上げる (raise) という挙動を示す．コントローラプロセス（図 1c）は，列車が踏切に近づいたらバーを下ろすように，また，列車が踏切を出たらバーをあげるように，遮断機に命令を発行する．

個々のプロセスの同期積 [5] による振る舞いの組み合せが，システム全体としての振る舞いである．LTS の場合，プロセス同士の並列結合 [6] によって，システムの振る舞いを表現する．図 1d に遮断機制御システムの全体の振る舞いを示す．2 つのプロセスの並列結合演算の概要は次の通りである．

- 2 つのプロセスに共通するイベントが付与された遷移については，両プロセスは同時に遷移する．この演算はプロセス間の同期機構である．例えば，列車プロセスとコントローラプロセスにおいて，approach と exit が共通するイベントであり，どちらのプロセスも状態 0 に位置する場合にのみ，共に状態 1 に遷移する．これは，図 1d の状態 0 から 1 への遷移に相当する．各プロセスの状態の順序対を，この同期機構による結合後の状態と考えることもできる．この見方の下では，図 1d の状態 0 と 1 は，それぞれ状態 $(0,0)$ と状態 $(1,1)$ とみる．

- 2 つのプロセスで共通には現れないイベントが付与された遷移については，各プロセスが任意の順序で交互に遷移する（インターリービングといわれる）．この演算は各プロセスの非同期実行に該当する．例えば，列車プロセスとコントローラプロセスの状態 1 からの遷移を考えると，イベント enter と lower は両プロセスで共有されないので，その順序は任意である．これは，図 1d の状態 1 から 4 への遷移のうち，2 を経由した場合と 3 を経由した場合によって表現される．上と同様に列車プロセスの状態とコントローラプロセスの状態の順序対を考えると，それぞれ，状態 $(1,1)$ から状態 $(1,2)$ を経由して状態 $(2,2)$ に達する場合と，状態 $(1,1)$ から状態 $(2,1)$ を経由して状態 $(2,2)$ に達する場合を表す．これは，イベント lower が発生する場合には，列車プロセスは同一状態にとどまってコントローラプロセスのみが遷移し，逆にイベント enter が発生する場合には，列車プロセスのみが遷移するとみることもできる．

この例のような，複数のプロセスで構成される並行システムは，システム全体としての挙動が複雑になり，分析や検証が難しい．本論文では，このような特徴を持つ

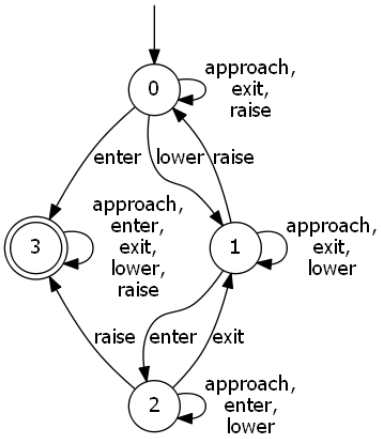


図 2. 性質の Büchi オートマトン

並行システムの検証を対象とする．図 1d のモデルを検証するために,「列車が踏切に入るならば，出ていくまで遮断機は下りている」という性質が成り立つかを考える．性質は時相論理式で形式的に記述されるが，オートマトンを用いたモデル検査では，論理式で書かれた性質を Büchi オートマトン [2] に変換する．Büchi オートマトンは受理状態を無限回通過する無限語を受理する有限オートマトンである．性質「列車が踏切に入るならば，出ていくまで遮断機は下りている」を表現した Büchi オートマトンを図 2 に示す．図 2 の二重丸は受理状態である．このオートマトンは，性質を満たさないイベント列を受理するように構成されている．例えば，遮断機のバーが下がる (lower) 前に列車が踏切に進入する (enter) イベント列は，状態 0 から 3 に遷移した後に，状態 3 を繰り返し通過するために受理されるが，性質に反する．

次に，図 1d と図 2 の並列結合によって，性質に違反するシステムのイベント列のみを受理するオートマトンを構成する (図 3)．このオートマトンの受理可能性を判定し，受理可能な場合には，受理するイベント列を反例として返す．受理可能性の判定には，受理状態を含む閉路探索を実行する．探索には，Tarjan のアルゴリズム [18] などの有向グラフの強連結成分 (SCC) の探索アルゴリズムや二重深さ優先探索法 [5, 2] が用いられる．SCC とは，与えられた有向グラフの部分グラフで，遷移を繰り返し辿ることで，その任意の 2 状態間が互いに到達可能であるものをいう．図 3 では，初期状態 0 から到達可能な受理状態 5，6，7，8，9 から成る SCC が存在する

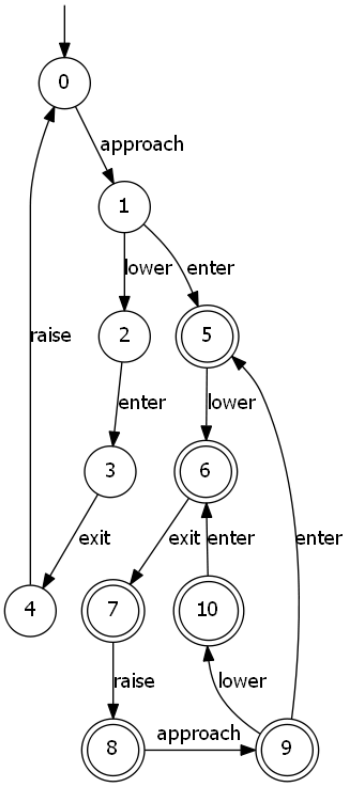


図 3. 並列結合オートマトン

ため，性質に違反すると判定される．よって，反例は，lower, exit, raise, approach, enter から成る閉路と，その閉路へのイベント列 approach, enter である．この反例は，遮断機制御システムには，遮断機のバーが閉まる前に列車が踏切に進入する恐れがあることを示している．

4. 行列計算に基づくモデル検査

本節では，提案する行列計算に基づいたモデル検査技法を詳しく述べる．また，その応用として，並行システムで問題となるデッドロック検出技法についても述べる．

4.1. 隣接行列による状態遷移の表現法

提案するモデル検査技法は，3 節で説明したオートマトン理論を用いたモデル検査に基づく．LTS や Büchi オートマトンを有向グラフとみなして，それぞれ隣接行列で表し，それらの演算によってアルゴリズムを構成する．なお，本節での行列の要素は真理値とみなすことと

し，要素の積と和は，それぞれ論理積と論理和とする．

隣接行列は状態間の遷移関係を記述した正方行列である．その (i,j) 要素は，状態 i から状態 j への遷移が存在する場合には 1，存在しない場合には 0 である．また，i 行目の行ベクトルは状態 i から出ていく遷移を，j 列目の列ベクトルが状態 j に入る遷移を表す．例えば，状態に与えた番号に 1 を加えた値を行番号として，イベントを無視すると，図 1a の隣接行列は次の通りである．

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

隣接行列を用いた検査アルゴリズムの特に重要な構成要素は，次の 3 点である．

1. LTS や Büchi オートマトンのようなイベントラベルのある有向グラフに対する隣接行列表現の構成法
2. 並列結合演算
3. 閉路の検出法と反例の生成法

第 1 の隣接行列表現の構成では，遷移に与えられているイベントの扱いが重要である．そこで，イベントごとに隣接行列を構成する．列車プロセス（図 1a）を例にとると，次の 3 つの行列は，左からそれぞれイベント approach，enter，exit についての隣接行列表現である．

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

列車プロセスにおけるイベント lower のようなプロセスに出現しないイベントについては，単位行列 I を行列表現とする．単位行列は，各状態が自身への遷移，つまり，自己閉路のみを持つことを意味する．これは，非出現イベントではプロセスが他の状態に遷移せず，同一状態にとどまることと解釈できる．

第 2 の並列結合演算は，それぞれのイベントについての隣接行列同士のクロネッカー積で実現する．ここで，$m_1 \times m_2$ 行列 $A = (a_{ij})$ と $n_1 \times n_2$ 行列 $B = (b_{ij})$ のクロネッカー積 $A \otimes B$ は次式で定義される $m_1 n_1 \times m_2 n_2$ 行列である (一般の定義は [19] に詳しい)．

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1m_2}B \\ \vdots & \ddots & \vdots \\ a_{m_1 1}B & \dots & a_{m_1 m_2}B \end{pmatrix} \qquad (1)$$

すなわち，$a_{ij}b_{kl}$ はクロネッカー積の $((i-1)n_1+k, (j-1)n_2+l)$ 要素である．このことは，状態 i から状態 j および状態 k から状態 l に共通イベントで遷移する場合，並列結合した LTS には状態 $(i-1)n_1+k$ から $(j-1)n_2+l$ の遷移が存在すると解釈できる．順序対 (i,k) は $(i-1)n_1+k$ と，順序対 (j,l) は $(j-1)n_2+l$ とそれぞれ 1 対 1 に対応するから，順序対で表した並列結合の状態 (i,k) から状態 (j,l) への遷移に対応するとみなしてよい．例として，図 1b と図 1c の LTS を考えると，両プロセスで共通するイベント lower に関しては，次のようにクロネッカー積を計算できる．

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$
$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

遮断機プロセスの状態 0 から状態 1 への遷移と，コントローラプロセスの状態 2 から状態 3 への遷移は，並列結合の状態 1 から状態 6 への遷移になることを意味する．

2 つの LTS に共通には現れないイベントの場合には，隣接行列の構成法から，式 1 の A または B は単位行列 I である．$A = I$ とすると，対角成分 $a_{ii} = 1$ で他は 0 なので，クロネッカー積 $A \otimes B$ の $((i-1)n_1+k, (i-1)n_2+l)$ 要素が b_{kl} で，他の要素は 0 である．これは，隣接行列 B で表されるプロセスのみが遷移する場合，すなわち，順序対で表した状態 (i,k) から状態 (i,l) への遷移に相当する．$B = I$ の場合も同様である．

SCC の検出を行う際には，探索空間に相当する有向グラフがあればよいので，各イベントを除いた単一の隣接行列を構成する．そのために，並列結合により得られた各イベントに対応する隣接行列の和を求める．

第 3 の閉路検出と反例生成については，行列計算を使用した探索技法である代数的グラフアルゴリズム [9] を使用する．モデル検査の実現のために，SCC の検出アルゴリズム [9] と幅優先探索法（BFS） [8, 9] を使用する．

SCC の検出アルゴリズム [9] は次の手続きから成る．並列結合オートマトン（図 3 参照）の n 次隣接行列を A，n 次単位行列を I として，$C = (A + I)^{n-1}$ を計算する．ここで，行列 $A + I$ の (i, j) 要素は，状態 i から状態 j に高々1 回遷移を辿って到達可能であれば 1，不可能であれば 0 である．同様に，C の (i, j) 要素は，状態 j が，状態 i から高々 $n - 1$ 回遷移を辿ることで到達可能ならば 1，不可能であれば 0 である．したがって，C は各状態から到達可能な状態の集合を情報として持つことに注意する．次に，C と C の転置行列 C^T の要素ごとの積（アダマール積）$S = C \circ C^T$ を計算する．C^T の (i, j) 要素は，状態 i が状態 j から到達可能ならば 1，不可能であれば 0 である．よって，行列 D は，状態 i と j が互いに到達可能な場合は 1 で，そうでない場合は 0 になるような (i, j) 要素から成る n 次対称行列である．S の各行ベクトルで，値が 1 の成分の集合が SCC を構成する．

検出した SCC に受理状態が含まれていた場合には，反例を生成する．反例は，検出した SCC と，初期状態からその SCC への路とで構成される．従来のモデル検査では，性能上有利であるとされる深さ優先探索（DFS）により探索が行われることが多い．しかしながら，隣接行列表現に対しては，BFS は行列の積により容易に実現が可能だが，DFS の実現は難しい．一方で，BFS で探索した路は最短路となるため，ユーザにとって理解しやすい反例を求めることができる [20]．そこで，初期状態から SCC への路を生成するために，SCC に含まれるいずれかの状態までの路を BFS により探索する．行列計算を用いた BFS の概要は次の通りである [9]．まず，探索の開始状態を行ベクトルで表現する．簡単のため，図 3 のように初期状態を 0 とすると，第一成分のみ 1 で他の $n-1$ 成分が 0 の単位行ベクトル $\boldsymbol{x}_0 = (1, 0, \ldots, 0)$ である．次に，$\boldsymbol{x}_1 = \boldsymbol{x}_0 A$ を計算する．$\boldsymbol{x}_1$ は，状態 0 から遷移を 1 回辿って到達可能な状態に相当する列のみ 1 となる行ベクトルである．同様に，i 回の探索結果 $\boldsymbol{x}_i$ に対して，$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i A$ を算出すると，$i+1$ 回遷移を辿ることで到達する状態の集合が得られる．探索の冗長性を除くために，$\boldsymbol{x}_{i+1}$ から i 回の遷移で既に到達した状態の集合を除いた後に，この手続きを繰り返せばよい．

4.2. デッドロックの検出

複数のプロセスから成る並行システムの場合には，プロセス間の相互作用の不備により，全てのプロセスが停止してしまうデッドロックが起こりうる．デッドロックは並行システムで発生する代表的な不具合の一つであり，多くのモデル検査系に検出機能が備わっている．ここでは，行列計算を用いたデッドロック検出技法を述べる．

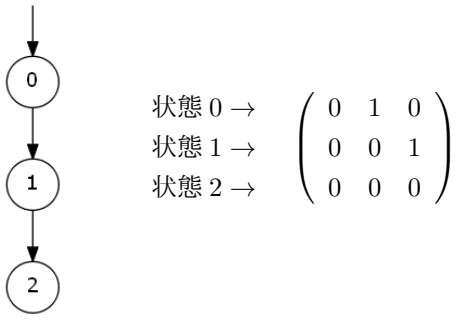


図 4. デッドロック状態（状態 2）のある LTS とその隣接行列表現

LTS で記述されたシステムでは，デッドロックは，各プロセスを並列結合した LTS においてイベントが発生しない状況を指す．よって，デッドロックの検証には，性質を記述した Büchi オートマトンは不要である．デッドロックを検出するには，初期状態から LTS の各状態を網羅的に探索して，出遷移のない状態（デッドロック状態）を検出する．隣接行列表現の場合には，第 i 行を取り出した行ベクトルが零ベクトルならば，対応する状態 i は出遷移が存在せず，デッドロック状態である．例えば，図 4 において，状態 2 はデッドロック状態である．隣接行列表現では，状態 2 に対応する第 2 行の行ベクトルが零ベクトルとなることがわかる．そこで，初期状態から BFS を実行して，到達した状態の行ベクトルが零ベクトルかどうかを判定し，零ベクトルならば，その状態でデッドロックが発生したと判断する．デッドロックを検出した際には，BFS で探索した初期状態からデッドロック状態への路を反例として返す．

5. 評価実験

提案手法の実現可能性を調べるために，Python 言語を用いて試作プログラムを実装し，評価実験を行った．Python は高性能な数値計算用パッケージが充実し，広く利用されていることから，実現可能性の検証に適していると判断した．Python のバージョンは 3.6.8 を利

用し，データ分析や機械学習の領域で広く使われている科学技術計算向けパッケージである SciPy [21] および NumPy [22] を使用した．

5.1. 評価用プロトタイプの実装

行列を扱うためのデータ構造には，NumPy の numpy.ndarray や numpy.matrix が知られている．これらのデータ構造は隣接行列の全要素の値を格納するので，状態数の 2 乗のオーダーのメモリが必要である．そのため，大規模な行列の計算には適していない．そこで，試作プログラムでは，SciPy の疎行列パッケージ scipy.sparse を用いた．疎行列とは，要素の多くが 0 であるような行列である．行列計算を効率化するために，scipy.sparse に実装されたデータ構造は，非零要素のみをメモリ上に保持する．したがって，隣接行列が疎行列ならば，scipy.sparse が適切なデータ構造である．実際，5.3 節で示すように，実験で使用したモデルは全要素数に対する非零要素数の割合が非常に小さいので，疎行列を使用するのは妥当であると考えられる．提案手法では，行列の要素の値は 0 または 1 であり，それらの値の演算には論理演算のみ使用するので，行列の値は Python の真理値型とした．また，並列結合演算を実現するために必要な疎行列のクロネッカー積は，scipy.sparse パッケージの演算ライブラリを使用した．

試作プログラムの入力は，システムを記述した LTS と，性質を記述した Büchi オートマトン，検証項目である．なお，システムが複数のプロセスで構成される場合には，それぞれを記述した LTS を入力とした．3 節で取り上げた遮断機制御システムの例の場合，入力する LTS は図 1a，図 1b，図 1c であり，Büchi オートマトンは図 2 である．LTS と Büchi オートマトンの入力には，LTS 用のデータフォーマットの一つである Aldebaran フォーマット [23] を用いた．検証項目は，性質の Büchi オートマトンを用いたモデル検査，あるいは，デッドロック検出のいずれかであり，それぞれ 4.1 節，4.2 節で述べた検査を実行する．反例を検出した場合は，試作プログラムは反例をイベント列で返し，そうでない場合は Python の組込み定数 None を返すこととした．

5.2. 評価に用いた例題

実験では，食事をする哲学者問題 [24] を採り上げた．この問題は，円卓を囲んで食事をする哲学者の人数に応じた数のプロセスから成る並行システムの排他制御を扱った問題である．文献 [24] や文献 [6] で具体的なモデルの例と共に詳しく議論されている．文献 [6] には，排他制御が正しく行われていないことが原因でデッドロックが発生する事例が，LTS の例と共に掲載されている．実験の際には，哲学者の人数を変更しながら，この事例の LTS をモデル検査系 LTSA [6] で生成した．その後，LTSA に備わっている Aldebaran フォーマットへの変換機能で，LTS を試作プログラムへの入力フォーマットに変換した．そして，提案手法を用いてデッドロックの検出を行った．図 5 と図 6 に使用した LTS を示す．図 5 は i 番目の哲学者プロセスで，席に着くと，左右の二つの食器を取って食事をした後に，立ち上がるという挙動を繰り返す．図 6 は補助プロセスである食器を表してい

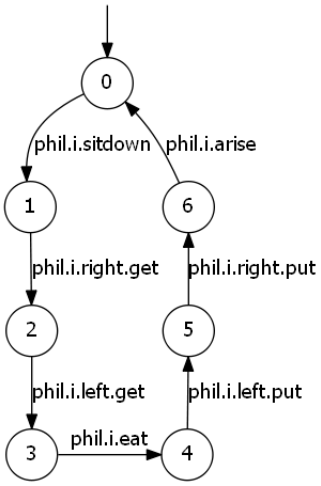


図 5. 哲学者 i プロセス

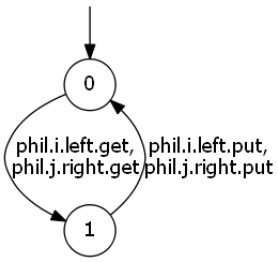


図 6. 食器 i プロセス．図中の j は，N を哲学者の人数として，$j = (i + 1) \mod N$ である．

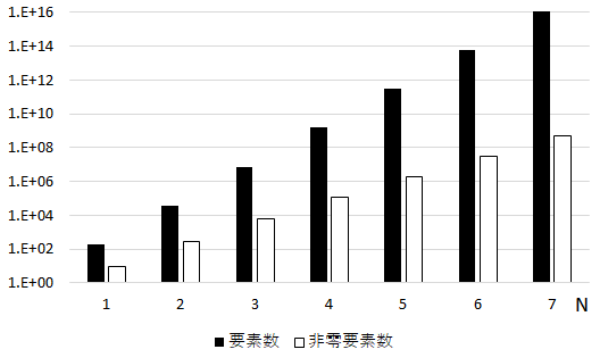


図 7. 哲学者の人数 ($N = 1 \ldots, 7$) に対する隣接行列の要素数と非零要素数

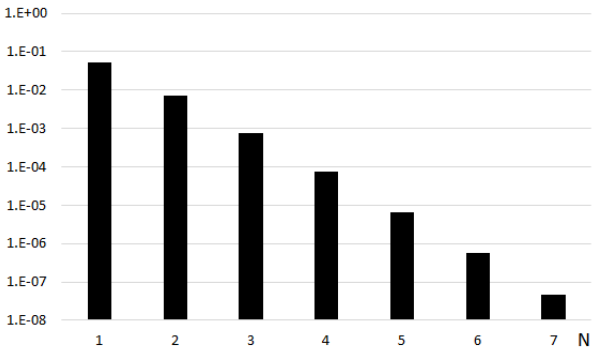


図 8. 哲学者の人数 ($N = 1 \ldots, 7$) に対する隣接行列の非零要素数の割合

る．食器は，その両側に座っている哲学者の一方にのみ使用される．本事例は，n 人の哲学者に対して，n 個の食器があるために $2n$ 個のプロセスで構成されている．

5.3. 評価結果 1: 行列表現の所要メモリ

最初に，試作プログラムで，食事をする哲学者問題の状態空間を隣接行列で生成して，その疎性を調べた．図 7 に，哲学者の人数を 1 から 7 まで変えた場合について，隣接行列の要素数と非零要素数を示す．図 8 には非零要素数に対する要素数の割合を示す．これらの結果から，人数が増加するにつれて非零要素の割合が指数関数的に減少しており，疎性が高くなる傾向を読み取ることができる．表 1 に，隣接行列とその疎行列それぞれについて，メモリ消費量の測定値を示す．隣接行列

表 1. 哲学者の人数 ($N = 1 \ldots, 7$) に対する隣接行列とその疎行列表現のメモリ消費量

N	隣接行列 [byte]	疎行列表現 [byte]
1	196	110
2	3.84×10^4	2.19×10^3
3	7.53×10^6	4.04×10^4
4	1.48×10^9	7.02×10^5
5	-	1.18×10^7
6	-	1.91×10^8
7	-	3.06×10^9

については numpy.ndarray を測定し，疎行列については圧縮行格納形式 scipy.sparse.csr_matrix を測定した．scipy.sparse.csr_matrix は，非零要素値の配列（data），非零要素の列番号を格納する配列（index），列番号格納配列において各行の開始位置を格納する配列（indptr）によって各要素を参照するので，それらの所要メモリの合計値とした．表 1 の「-」は，メモリ不足のために Python の例外 MemoryError が発行され，測定ができなかったことを示す．表 1 からも，食事をする哲学者問題では，疎行列が妥当なデータ構造であることがわかる．

5.4. 評価結果 2: 所要時間と実行時消費メモリ

次に，試作プログラムを用いて，哲学者の人数 N を 1 から 7 まで変更しながら，食事をする哲学者問題に含まれるデッドロックの検出を行い，実行時間と検証実行時の最大消費メモリ量を測定した．実験で使用した計算機は，Intel Core i7-8700，CPU 3.20GHz，RAM 16.0 GB を備えた HP ProDesk 600 G4 SFF である．また，OS には Windows 10 Pro を使用した．

表 2 に提案手法の試作プログラムが検出した反例の長さ，並びに試作プログラムの性能を測定した結果を示す．同じ表に，同じ LTS のデッドロックを LTSA で検出した結果も示す．どちらも 5 回実行した平均値をとった．LTSA はスタンドアローンで動作するモデル検査系のため直接性能を測定することが困難である．そのため，結果については，実行時のログに出力された解析時間とメモリ消費量から算出した．本実験の目的は性能の比較ではなく，提案手法の実現可能性の評価なので，LTSA の結果は参考値である．

表 2. 哲学者の人数 ($N = 1 \ldots, 7$) に対する提案手法と LTSA の反例の長さと性能の測定結果

	提案手法			LTSA		
N	反例長	実行時間 [sec]	最大メモリ消費量 [MiB]	反例長	実行時間 [sec]	最大メモリ消費量 [MiB]
1	2	0.042	93.1	2	0.0002	11.4
2	4	0.092	93.7	4	0.000	11.5
3	6	0.154	94.4	6	0.000	11.8
4	8	0.154	100.0	8	0.0002	12.8
5	10	0.624	182.1	10	0.0014	14.1
6	12	5.317	1547.8	12	0.008	21.1
7	14	2968.4	2232.1	14	0.063	24.2

実験の結果，実験したすべてのモデルについて，デッドロックの検出に成功した．また，LTSA と同様に，デッドロックに至る最短のイベント列を反例として返した．例として，$N = 3$ の場合について，反例を次に示す．

phil.0.sitdown
phil.0.right.get
phil.1.sitdown
phil.1.right.get
phil.2.sitdown
phil.2.right.get

哲学者は phil.0，phil.1，phil.2 の 3 人である．上の反例は，各哲学者が席に着いて自分の右側の食器を取ったところで食卓に食器がなくなり，デッドロックに陥ることを表している．N が他の値の場合でも，検出した反例は同様の手順でデッドロックとなった．

最後に，性能評価の結果を論じる．表 2 から，現状の試作プログラムは，実行速度，実行時メモリ消費量のいずれの効率性の点でも LTSA には及ばないことがわかる．特に，$N = 7$ になると，提案手法の実行時間が急激に悪化している．これは，主として並列結合の性能の問題である．クロネッカー積によって並列結合を実現すると，モデル検査では不要な，初期状態から到達不可能な状態も生成されるため，行列の規模が巨大化する．例えば，3 節で扱った踏切の遮断機制御システムでは，提案手法で構成される並列結合オートマトンの実際の隣接行列の次数（状態数）は，クロネッカー積の定義から，図 1a，図 1b，図 1c，図 2 の状態数の積 96 となるが，図 3 より初期状態から到達可能な状態数は 11 である．本実験の場合，$N = 7$ のときの隣接行列の次数が約 1×10^8 と大規模となり，計算機の CPU の性能では扱うことが困難になったと考えられる．このことは，モデル検査の主要な研究課題である状態爆発問題 [2] が発生したと解釈できる．試作プログラムには状態爆発問題への対策されていないため，$N = 7$ で性能の劣化が著しくなったと思われる．今後，さらに大規模なシステムとその隣接行列を扱えるようにするためには，使用するデータ構造を工夫する，到達不可能な状態を除去する機構を実装する，などの対処をする必要がある．

6. おわりに

本論文では，線形代数，特に行列計算を用いてモデル検査技法を構成した．従来のオートマトン理論を用いたモデル検査は，有向グラフの探索技術に重点を置いている．本論文では，グラフを隣接行列で表すことで，行列演算を用いて検査に必要なアルゴリズムを構築した．モデル検査は大規模なグラフから成る探索空間を扱うことが多いが，機械学習などの技術の進展とともに，大規模行列の演算を計算機で実行できるようになってきた．また，安価な GPU の普及により，行列演算を用いたモデル検査の更なる高性能化も期待できる．デッドロック検出問題を対象として，提案手法の実現可能性について評価実験を実施した結果，実験に使用した問題に含まれるデッドロック，並びに，既存のモデル検査系と同程度の長さの反例を検出することができた．したがって，提案手法は実現が可能であると考えられる．一方で，性能面では，状態爆発が発生し，既存のモデル検査系には及ばないという結果となった．

今後取り組むべき課題は二点ある．第一に，Büchi オートマトンを使ったモデル検査の評価実験を行う．この場合，強連結成分の検出を行う必要があるので，計算時間

とメモリ消費量の双方がデッドロック検出よりも増大することが予想される．第二に，検査性能の改善を行う必要がある．本論文の評価実験は CPU 上での実現性の確認にとどまっている．そこで，GPU を搭載した計算機や分散環境での検査の実現に向けて，並列化などの高性能計算分野の成果 [25] を活用を進める．GPU 上で実行する On-the-fly 検査法 [12] や半順序簡約法 [13] などのモデル検査の最適化技術を行列計算に取り入れる．クロネッカー積の効率的な計算には，文献 [26] で研究されている科学技術計算向けの汎用高速計算技法を組込むことも有効であると考えられる．

謝辞

有益なご指摘を頂いた査読者の方々に感謝する．

参考文献

[1] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1981.

[2] Edmund Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT Press, 2nd edition, 2018.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] Moshe Y. Vardi and Pierre Wolper. An automata theoretic approach to automatic program verification. In *Proceedings of First Symposium on Logic in Computer Science*, pages 332–344, 1986.

[5] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[6] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., 2nd edition, 2006.

[7] 櫻井鉄也. 巨大行列はなぜ重要か. **数学セミナー**, 59(2):8–11, 2020.

[8] 藤澤克樹. 巨大行列とグラフ解析. **数学セミナー**, 59(2):24–28, 2020.

[9] Kayhan Erciyeş. *Guide to Graph Algorithms: Sequential, Parallel and Distributed*. Springer, 2018.

[10] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266, 2003.

[11] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Češka. Designing fast LTL model checking algorithms for many-core GPUs. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097, 2012.

[12] Ezio Bartocci, Richard DeFrancisco, and Scott A. Smolka. Towards a GPGPU-parallel SPIN model checker. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 87–96, 2014.

[13] Thomas Neele, Anton Wijs, Dragan Bošnački, and Jaco van de Pol. Partial-order reduction for GPU model checking. In *Automated Technology for Verification and Analysis*, pages 357–374, 2016.

[14] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science*, volume 1579, pages 193–207, 1999.

[16] Anton J. Wijs and Dragan Bošnački. Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In *Model Checking Software*, pages 98–116. Springer Berlin Heidelberg, 2012.

[17] Pedro Rodrigues, Emil Lupu, and Jeff Kramer. LTSA-PCA: Tool Support for Compositional Reliability Analysis. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 548–551, 2014.

[18] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[19] 佐武一郎. **線型代数学**. 裳華房, 1974.

[20] Viktor Schuppan and Armin Biere. Shortest counterexamples for symbolic model checking of LTL with Past. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 493–509, 2005.

[21] SciPy: https://www.scipy.org/.

[22] NumPy: https://numpy.org/.

[23] Aldebaran Format: https://www.mcrl2.org/web/user_manual/language_reference/lts.htm.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[25] Ilja Zakharov. A survey of high-performance computing for software verification. In *Tools and Methods of Program Analysis*, pages 196–208. Springer, 2018.

[26] Paul L. Fackler. Algorithm 993: Efficient computation with kronecker products. *ACM Transactions on Mathematical Software*, 45(2), 2019.