

余因子展開の性質を利用した動的検証の高速化

飯田 隆一
岡山大学

pq425f9y@s.okayama-u.ac.jp

中川 博之
岡山大学

h-nakagawa@okayama-u.ac.jp

要旨

自己適応化されたシステムは、環境の変化に応じてシステムが自身を検証し、必要に応じて振る舞いを変更することができる。自己適応化の検証段階を実現する手法として Filieri らの検証式を用いる方法があるものの、この方法では実行中に検証式の再生成が必要となった際に計算コストが増大する。この課題の解決策として、キャッシングにより中間式を再利用する手法がある。しかし、システムモデルのサイズ増大に伴う計算時間の増加が問題となる。本研究では、先行研究において重要な処理である余因子展開の性質に着目し、特に巨大なモデルにおける検証時間の抑制を目指す。また、検証式再生成を想定した実験を行い、計算時間とメモリ使用量を比較したところ、状態数の多いモデルに対しては、提案手法がより高速に計算できることを確認した。

1 はじめに

ソフトウェアシステムは社会の様々な領域で利用されるようになり、あらゆる状況下で障害なく動作することが求められている。しかし、様々な状況変化をあらかじめ予測して開発するアプローチは、組み合わせの爆発による計算コストの増大、およびそれにとまなう条件分岐の複雑化が生み出す開発・運用リスクから現実的ではない [1]。

この問題を解決する方法として、システムが環境変化に際して自己検証を行い、必要に応じて自律的に振る舞い (behavior) を変更することのできる自己適応化 (self-adaptation) [2][3][4] がある。自己適応化において、システムが自身を検証するための手法として Filieri ら [1] の検証式生成法が知られている。この手法では変化しうる

状況を変数として定義し、変数とシステム要求から生成される検証式を生成する。実行時観測されるパラメータを検証式に代入することで、システム要求の検証がより低い計算コストで実行できる。これにより動的検証の効率化が可能である。

しかしこの手法では、変数の代入では対応できない大規模な振る舞い変更が起こった場合、システムの実行中に検証式を再生成する必要がある。十分な計算機資源を見込める設計時とは異なり、システムの実行時においては、計算資源が限られていることが多い。そのため、再生成に要する計算コストの増大により、自己適応システムに期待される環境への適応性と Filieri らの手法に期待されるリアルタイム性のどちらかを損なう可能性がある。先行研究 [5] では、検証式生成時の中間式をキャッシュに保存し、再生成時に再利用することで計算時間を削減する手法を提案している。

本研究では、既存手法で用いられる再帰的な余因子展開のもつ性質に着目し、検証式の再生成に必要な行列式計算の削減を目指す。本論文では、2 節で研究背景として確率的モデル検査を用いた動的検証の手法、および先行研究であるキャッシュを用いた検証式再生成手法とその問題点を説明する。3 節では提案手法である再帰的な余因子展開の性質を用いた問題点への取り組みを、4 節では提案手法を有効性検証のための実験について述べる。最後に実験結果についての考察を 5 節で述べた後、6 節で結論を述べる。

2 研究背景

2.1 自己適応システム

自己適応システムとは、環境の変化に対して自動で振る舞いを変更するシステムである。システムが環境の変

化を検知・分析し、自身に適切な振る舞い変更を行うことにより、システムの大規模化や想定環境の多様化に伴う開発・運用コストの増大を抑制できる。自己適応システムを実現する構成概念の一つに MAPE-K フィードバックループ [6] があり、監視 (Monitor)・分析 (Analyze)・計画 (Plan)・実行 (Execute) から成る 4 つのステップと、ステップ間で共有される知識 (Knowledge) から構成される。自己適応システムの監視ステップでは外部環境やシステム内部の情報を取得する。次の分析ステップでは、監視ステップで得た情報から、現在のシステムが要求を満たしているかを検証する。続く計画ステップでは、要求達成のためのシステムの振る舞いを決定する。最後の実行ステップでは、計画に従いソフトウェアシステムを変更する。各ステップで情報を共有しながら繰り返すことで、変化する環境に適応可能なシステムを実現する。分析ステップの実現には、システムが要求を満たしているかを実行中に判断する動的検証の技術が有効である。自己適応システムの動的検証に適した手法として、Filieri らの確率的モデル検査を用いた手法 [1] がある。

2.2 Filieri らによる確率的モデル検査を用いた動的検証手法

モデル検査とはアルゴリズムやシステムの設計に対してその状態空間を探索することで、性質が満たされるかどうかを調べる検証手法であり [7]、特にシステム要求の可到達性について検証するために確率的モデル検査が用いられる。確率的モデル検査では、システムの振る舞いを数理モデルに変換し、システムに対する要求を時相論理式として記述することで、システムの可到達性を数理的に検証できる [8]。一方で、モデル検査は通常、設計時にモデルを作成することを前提とした手法であるため、設計時に想定していない環境の変化が実行時に発生した場合、システムを実行しながら新たな振る舞いを反映したモデル検査器を再生成する必要がある。よってそのための計算資源と実行時間が課題となる。

Filieri らの手法では、実行時のモデル再生成に必要な計算の一部を設計時に分担することで、実行時の計算資源と実行時間の低減を図っている。具体的には、設計時に想定しうる環境変化を状態空間上の変数として定義し、変数とシステム要求からモデル検査器に相当する検証式を作成する。検証式は実行時に監視ステップで得られたパラメータを変数に代入することで可到達性を検証でき

一般的確率的モデル検査



一般的確率的モデル検査

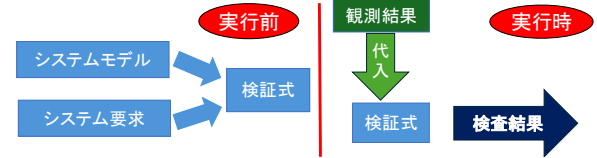


図 1. 一般的な確率的モデル検査手法と Filieri らの手法の比較

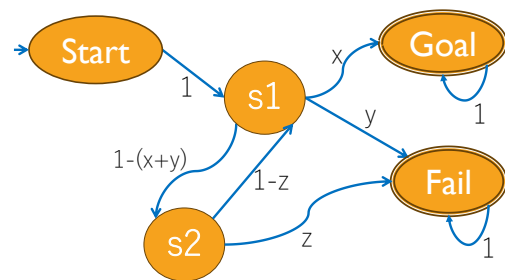


図 2. DTMC モデルの例

るため、実行時間や計算資源の観点において効率的な確率的モデル検査を実現している。図 1 に、一般的な確率的モデル検査手法と比較した Filieri らの手法の概略を示す。一般的な確率的モデル検査ではシステムモデルの構築とシステム要求からモデル検査器を作成するため、環境の変化によりシステムの振る舞いが増えた場合は実行中にすべての処理を再実行する必要があったが、Filieri らの手法では検証式に代入する値を変更するだけでよい。

Filieri らの研究 [1] では、システムのモデル化に離散時間マルコフ連鎖 (Discrete Time Markov Chains:DTMC) を用いている。これは、システムの状態集合を離散時間状態の一次マルコフ過程としてモデル化するものである。例として、簡単なシステムの振る舞いを DTMC モデルに表現したものを図 2 に示す。このシステムでは、初期状態 *Start* のほかに遷移状態 *s1* と *s2* があり、可到達性評価のために吸収状態として成功処理 *Goal* と失敗処理 *Fail* を設定している。状態間の遷移確率が矢印上に表記されており、遷移確率上で定義される変数 x, y, z はシステム実行時に値が決定するパラメータである。

DTMC モデルはそれぞれの状態間で遷移する確率を表

	Start	S1	S2	Goal	Fail
Start	0	1	0	0	0
S1	0	0	$(1-x-y)$	x	y
S2	0	$(1-z)$	0	0	z
Goal	0	0	0	1	0
Fail	0	0	0	0	1

図 3. 遷移行列の例

現しているため、DTMC モデルの情報は状態遷移行列に変換することができる。状態遷移行列とは異なる状態間の遷移確率を表現する行列であり、状態遷移行列を用いることでシステムの振る舞いを数理的に処理することができる。例として、図 2 から変換できる状態遷移行列を図 3 に示す。この行列では初期状態 *Start* から 1 ステップ後に遷移状態 *S1* に到達する確率は、1 行 2 列目の要素である 1 であり、遷移状態 *S1* から 1 ステップ後に吸収状態 *Goal* へ到達する確率は、2 行 4 列目の要素である x である。この状態遷移行列を P とおくと、2 ステップ後の状態遷移行列は P^2 で求めることができる。同様に、 n ステップ後の状態遷移行列は P^n である。

可到達性評価のためには、実行時のある状態からいずれかの吸収状態への最終的な到達確率を求める必要がある。 $i \neq j$ のとき、現在の状態 i から最終的に吸収状態 j に到達する確率は、行列 P^∞ の ij 要素として求めることができる。これを求めるため、遷移行列 P を要素の性質によって以下の 4 行列に分解する。

$$P = \begin{pmatrix} Q & R \\ O & I \end{pmatrix}. \quad (1)$$

行列 Q は遷移状態間の遷移確率を表す状態遷移行列である。行列 R は遷移状態から吸収状態への状態遷移行列である。行列 O は吸収状態から遷移状態への状態遷移行列だが、吸収状態の定義からゼロ行列となる。行列 I は吸収状態間の状態遷移行列だが、同様に吸収状態の定義から単位行列である。例として、図 3 を分解した行列を図 4 に示す。

このような分解により、最終的な到達確率 P^∞ 以下のように表される。

$$P^\infty = \begin{pmatrix} Q^\infty & N \times R \\ O & I \end{pmatrix}. \quad (2)$$

ただし、 N は以下のように定義される [9]。

	Start	S1	S2	Goal	Fail
Start	0	1	0	0	0
S1	0	0	$(1-x-y)$	x	y
S2	0	$(1-z)$	0	0	z
Goal	0	0	0	1	0
Fail	0	0	0	0	1

図 4. 行列 P を分解した例

$$N = \sum_{k=0}^{\infty} Q^k. \quad (3)$$

ここで行列 Q について、各行の要素の合計が 0 以上 1 未満、すなわち行列 Q の一様ノルムが 1 未満であるため、

$$\lim_{n \rightarrow \infty} Q^n = O. \quad (4)$$

したがって式 2、式 3、式 4 から、

$$P^\infty = \begin{pmatrix} O & \sum_{k=0}^{\infty} Q^k \times R \\ O & I \end{pmatrix}. \quad (5)$$

この行列は現在の状態 i から目標とする状態 j への最終的な到達確率を i 行 j 列に格納した行列である。 $B = N \times R$ とおくと、特に遷移状態 i から吸収状態 j への最終的な到達確率を求めるには B の ij 要素を求める必要がある。

式 3 について、行列 N は行列 Q の無限等比級数であるため、

$$\lim_{n \rightarrow \infty} N = (I - Q)^{-1}. \quad (6)$$

したがって、行列 $I - Q$ の逆行列として求めることができる。ここで、行列 $W = (I - Q)^{-1}$ について、 W の ij 要素 w_{ij} に対する余因子を $C_{ij}(W)$ とおくと、行列 N の ij 要素 n_{ij} は以下ようになる。

$$n_{ij} = |W|^{-1} \times C_{ji}(W). \quad (7)$$

行列 N は正方行列であるから、その大きさを t とすると、遷移状態 i から吸収状態 j への最終的な到達確率 b_{ij} は、 $B = N \times R$ より、以下のように求められる。

$$b_{ij} = \sum_{k=1}^t |W|^{-1} C_{ki} \times r_{jk}. \quad (8)$$

これにより導かれる、変数を含む多項式を検証式 (verification formulae) と呼ぶ。設計時に計算された検証式に対し、実行時にパラメータを代入することで環境の変化に対応したモデル検査器として機能させることができる。

以上のように、Filieri らの手法ではシステムモデルに変数を導入した検証式を利用することで、システムの実行時における環境変化への高速な適応を実現した。しかし、変数によって適応できる環境変化の範囲には限界がある。変数によって定義できる範囲を超える大規模な環境変化やそれに伴うシステムの振る舞い変更が発生すると、設計時に作成した DTMC モデルや検証式が利用できなくなるため、新たな検証式の生成が必要になる。そのため、自己適応システムが適応範囲を拡大するためには、システムの走行中に検証式の再生成を高速に行う必要があり、特にボトルネックとなっている $|W|$ の計算を高速化する手法が求められていた。

2.3 Fujimoto らのキャッシュを用いた検証式再生成手法

行列式 $|W|$ を高速に計算する手法として Fujimoto らの手法 [5] がある。この手法では行列式計算のボトルネックとして余因子展開が注目されている。余因子展開（または Laplace 展開）とは、行列を分解し、ある行や列の各要素とその要素に対する小行列（部分行列）の一次結合/線形和に変換する操作である。大きさ n の正方行列 A について任意の行 i を選んだ時、第 i 行のそれぞれの成分 $a_{ij} = a_{i1}, a_{i2}, a_{i3}, \dots, a_{in}$ に対する余因子を C_{ij} と置くと、以下の展開式が得られる。

$$|A| = a_{i1}C_{i1} + a_{i2}C_{i2} + a_{i3}C_{i3} \dots + a_{in}C_{in} = \sum_{j=1}^n a_{ij}C_{ij}. \quad (9)$$

展開された各項の行列式に対して、再帰的に余因子展開を適用することで行列式を計算することができる。変数を含む行列に対しても行列式計算が可能のため、Filieri らの研究等の先行研究でも広く用いられている。一方で必要な計算コストが高く、同じ処理に必要な余因子展開の実行回数を低減することはプログラム全体の処理速度に影響する課題であった。そのために先行研究 [10] では、繰り返される余因子展開の過程で出現する行列式と個々の計算結果をキャッシュとして保存し、実行時の余因子展開において同じ行列式が出現した際（キャッシュヒット）に保存した解を再利用する手法を提案している。これにより実行時に必要となる余因子展開を設計時に分担することが可能である。

Fujimoto らの手法 [5] はこのアプローチにおける発展的手法の一つである。Fujimoto 手法では、候補行列の作成とモデルの差分情報を用いた最適化により、キャッシュヒット率が向上している。Fujimoto 手法の概略図を図 5 に示す。

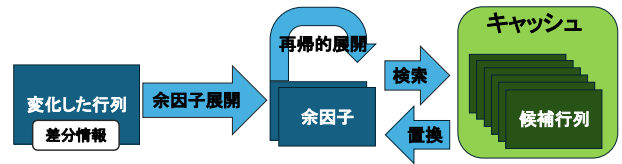


図 5. Fujimoto 手法の概略図

$$\begin{pmatrix} 0 & 1-x-y & 0 & x & y \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \xrightarrow{\text{余因子展開}} = - \begin{pmatrix} 0 & (1-x-y) & 0 & x & y \\ 0 & \alpha & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} - (1-x-y) \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & (1-z) & z & 0 & 0 \\ 0 & 0 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} + \dots = (1-z) \begin{pmatrix} 0 & (1-x-y) & 0 & x & y \\ 0 & \alpha & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} - \dots$$

図 6. 候補行列生成の例

自己適応システム全体の規模と比較した場合、環境の変化に伴って発生する変更は部分的なものであると仮定できる。変化の前後でシステムの大部分が共通するため、その共通部分に対応する行列式計算であれば設計時にあらかじめ計算・保存しておくことが可能であるため、キャッシュによる実行時の計算時間短縮が可能となる。一方、設計時においてどの部分の振る舞いに変化するかを予測することはできないため、Fujimoto 手法では設計時の遷移行列に余因子展開を繰り返すことで、環境変化の後であっても再利用できる可能性の高い行列を列挙している。図 6 に例を示す。Fujimoto 手法では大きさ n の遷移行列に対し、行方向に n 回、列方向に n 回のラプラス展開を行っている。

このように展開された各行列のそれぞれに対し行列式を計算し、各行列のハッシュ値と行列式計算により得られた多項式の組をキャッシュとして保存する。図 7 に例を示す。行列のハッシュ値がキー、行列式計算の結果を値としたハッシュテーブルとしたレコードをキャッシュを保存する処理を、すべての候補行列に対し行っている。

実行中に振る舞いの変更により検証式を再生成する際、Fujimoto 手法ではモデルの差分情報を用いた余因子展開を行い、得られた行列に対してキャッシュ検索を行う。検索の結果キャッシュヒットした場合、その行列を保存されている多項式と置き換えることで該当部分の行列

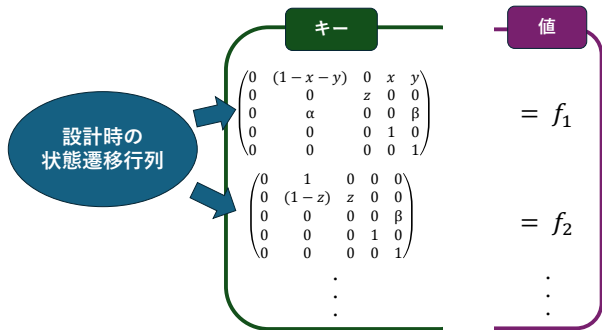


図 7. キャッシュ保存の例

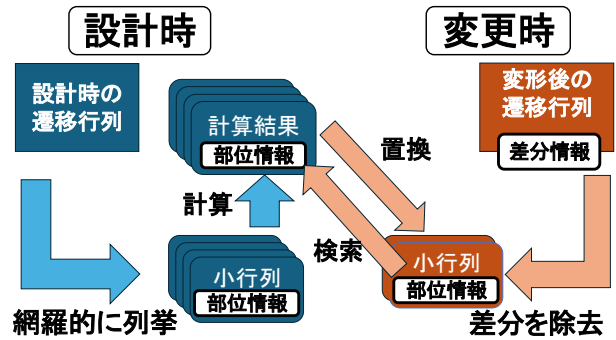


図 9. 提案手法の概要図

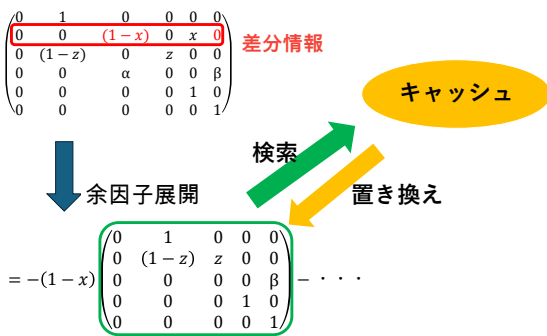


図 8. モデル差分情報に基づいた余因子展開とキャッシュ検索の例

式計算を省略する。ここで、差分情報とはシステムの振る舞いに変更した際の変更箇所の情報である。変更が行われた行や列の箇所を特定し、その行・列から余因子展開を行うことで、変更後の行列から変更された箇所を除去できるため、残った行列はキャッシュ内に保存された行列と一致しやすくなる。展開された行列に対しキャッシュ検索を行い、すべての行列が置き換えられなかった場合、残った行列に対して再帰的に余因子展開と検索を行う。図 8 に例を示す。この例では、2 行 3 列目の要素と 2 行 4 列目の要素が変化するという情報をもとに、2 行目に対して余因子展開を行い、展開された各行列式に対してキャッシュ検索を行う。

Fujimoto 手法は、振る舞い変更の影響が部分的な場合に高速な検証式の再生成を行う。しかしこの手法では、他のキャッシュを用いた先行研究 [11][10] と同様に、余因子展開によって得られる行列が必ずしもキャッシュヒットするとは限らないため、すべての行列が置き換えられるまで計算コストの高い余因子展開を繰り返す必要があ

る。そのためキャッシュによる余因子展開の回数削減には限界がある。また、余因子展開による行列式計算の処理は、行列のサイズに対して指数関数的に増大することが知られており、そのため大規模なモデルに対する動的検証では計算時間上の課題がある。既存の研究では余因子展開の回数削減により計算時間の低減に成功していたが、巨大な状態遷移行列に対する実行時間の指数関数的な増大を解決することができていない。そのため、実行時の計算時間を削減する手法が求められている。

3 提案手法

3.1 概要

本節では、前節で述べた問題点を踏まえ、実行時の検証式再生成を最適化することで計算時間を削減する手法を提案する。提案手法では、Fujimoto 手法における行列式計算の再利用と、変化前後での差分情報を用いる考え方を元に、再帰的な余因子展開処理の持つ性質に着目し、状態の追加や削除が無い場合の計算時間を削減する。

提案手法の概要図を図 9 に示す。図に示すように提案手法では、状態遷移行列から展開される小行列をその部位情報によって管理する。部位情報とは、小行列を構成する行・列についての情報であり、展開前の行列のどの部位からなる小行列かを記述している。変更された状態遷移行列に対し、差分情報を利用して差分を除去した小行列の組になるよう余因子展開を行う。展開された小行列は変更前の状態遷移行列と共通する部分のみから構成されるため、その部位情報を用いて保存された計算結果と置き換える。

3.2 再帰的な余因子展開処理が持つ性質

2.3 節で示したように、既存手法において余因子展開は検証式生成において重要な役割を持つと同時に、計算コストの高い処理である。既存手法では余因子展開によって展開された各項の行列式について再帰的に余因子展開が適用される。

正方行列 A に対して、 A の第 i 行と第 j 列を取り除いた行列を A_{ij} とすると、展開される行列式は、式 9 内の余因子 C_{ij} について以下のように表される。

$$C_{ij} = (-1)^{i+j} \times A_{i,j}. \tag{10}$$

したがって余因子展開の一般式は、以下のように表すことができる。

$$A = \sum_{j=1}^n a_{ij} C_{ij} = \sum_{j=1}^n (-1)^{i+j} \times a_{ij} \times A_{ij} \tag{11}$$

ここで A_{ij} についての定義を拡張し、 A の第 i 行と第 k 行、第 j 列と第 l 列を取り除いた行列を $A_{(i,k)(j,l)}$ とする。次に、式 11 式に対し、 $k(\neq i)$ 行目で余因子展開した場合、以下の式を得る。

$$A = \sum_{j=1}^n \sum_{l=1}^{n-1} (\pm 1) \times a_{ij} \times a_{kl} \times A_{(i,k)(j,l)} |_{j \neq l}. \tag{12}$$

(± 1) の符号については i, j, k, l の大小関係によって変化するため、ここでは考えないものとする。

続いて、行列 A を k 行目で余因子展開した後に l 行目で余因子展開した場合を考える。同様に考えると、

$$A = \sum_{l=1}^{n-1} \sum_{j=1}^n (\pm 1) \times a_{kl} \times a_{ij} \times A_{(k,i)(l,j)}. \tag{13}$$

ここで、12 式で得られた小行列と 13 式で得られた小行列を比較すると、 $A_{(ik|jl)}$ は行列 A の i 行、 k 行、 j 列、 l 列を除いた小行列であり、 $A_{(ki|lj)}$ は行列 A の k 行、 i 行、 l 列、 j 列を除いた小行列であるため、この 2 つの小行列は等しい。

帰納的に考えると、行 i, k, m, \dots について再帰的に展開した場合、得られる小行列は、展開した行の順序に関係なく、 $A_{(i,k,m,\dots)(j,l,n,\dots)}$ になる。これは列について展開した場合や、行と列の双方について再帰的に展開した場合も同様である。具体例を図 10 を用いて説明する。図内の行列 A は 6 行 6 列の正方行列である。再帰的な余因子展開の結果、2,3,6 行目と 2,4,5 列目が除去された小行列 $A_{(2,3,6)(2,4,5)}$ が得られたとする。小行列の符号を無視すると、この時、除去された結果の小行列は、行・

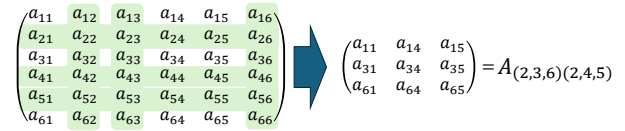


図 10. 符号を無視した場合の展開された小行列の概念図

$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	1列目を除去	2列目を除去	3列目を除去
1行目を除去	$\begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	$\begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix}$	$\begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$
2行目を除去	$\begin{pmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{pmatrix}$
3行目を除去	$\begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$

図 11. 符号を無視した小行列分類の例

列を除去する順番に影響しない。2 行 2 列目・3 行 4 列目・6 行 5 列目の順番に除去された場合も、2 行 5 列目・3 行 2 列目・6 行 2 列目の順番に除去された場合も得られる小行列は同じである。

この性質を利用することで、再帰的な余因子展開から得られた小行列は、どの行・列が除去されたかという情報によって分類・場合分けすることができる。具体例を図 11 に示す。この図では除去する行・列に注目することで、大きさ 3 の正方行列 A を余因子展開したときに得られるすべての小行列を表している。より大きな正方行列に対しても、同様に場合分けすることですべての小行列を列挙することができる。

3.3 提案手法

提案手法では、3.2 節で示した性質を利用し、余因子展開によって得られる小行列を部位情報によって管理する。部位情報とは、その小行列がどの行・どの列が取り除かれることで得られたかを記述する情報である。3.2 節で示した図 10 の具体例では、右辺の小行列を $A_{(2,3,6)(2,4,5)}$ として識別した。このとき、元の行列から 2 行目・3 行目・6 行目、および 2 列目・4 列目・5 列目が取り除かれているため、取り除かれた行・列を 0、残った行・列を 1 とする 2 進数表現で、行と列の部位情報は $((100110), (101001))$

と表される。部位情報に基づいた場合分けにより、状態遷移行列から得られるすべての小行列をあらかじめ列挙することができる。提案手法は以下の流れで実行される。

1. 設計時に、遷移行列から展開されるすべての小行列の部位情報と対応する行列式の計算結果を保存する。
2. 実行中に状態遷移行列が変化した際には、元の行列との差分情報を取得する。
3. 余因子展開により差分を除去し、残った小行列の部位情報を用いて保存された行列式と置き換える。

既存手法 [5] のキャッシュに相当する連想配列を生成する処理について説明する。設計時のモデルから得られる状態遷移行列から、部位情報を用いて場合分けを行い、考えられる小行列すべてを列挙する。得られた小行列全てに対して行列式計算を行い、部位情報をキー、対応する小行列の行列式計算結果を値とするハッシュテーブルを生成し保存する。

次に、実行中の環境変化によってシステムの振る舞いに変化し、システムモデルの変更が起きた場合の処理について説明する。本研究では、状態の追加や削除がない場合を考え、状態遷移行列内の要素変更に着目する。まず Fujimoto 手法 [5] と同様に、変更されたモデルの差分情報を取得する。モデルの差分情報とは、システムの振る舞いに変更した際のモデルの変更箇所の情報である。特に本研究では、状態遷移行列内のどの要素が変更されたかという情報を示す。システムの振る舞いに変化したとき、設計時の状態遷移行列と変更された状態遷移行列を比較し、どの要素が変更されたかについての情報を取得した後、余因子展開によって変更された要素を行列から取り除く。このとき得られる小行列は、変更された要素を含む行・列を取り除いたものであるため、変更前の状態遷移行列において対応する行・列を取り除いた場合に得られる小行列と同一である。具体例を図 12 に示す。この例では、変更された要素が 2 行目にあるため、余因子展開によって 2 行目と 3 列目を除去して得られる小行列 A_{23} は、設計時に同じ部分を除去して得られる小行列 A_{23} と等しくなる。これは、2 行目を除去して得られるほかの小行列に関しても同様である。

これにより、変更された要素を取り除いた小行列は、同じ部位情報をもつ設計時の小行列と等しくなる。したがって、得られた小行列の部位情報についてキャッシュ検索することで、対応する行列式の計算結果が得られる。

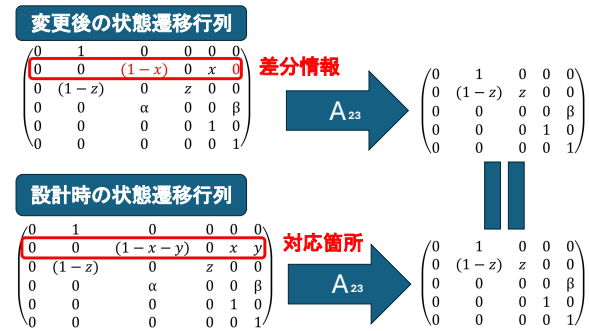


図 12. 部位情報が対応していることの例

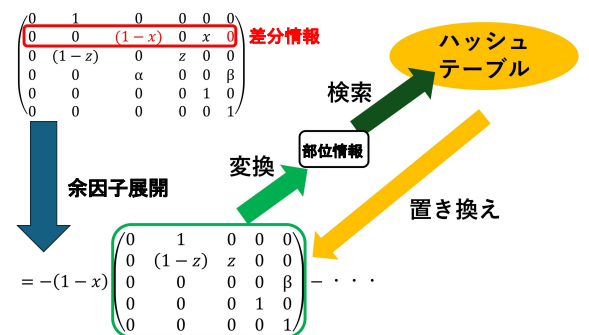


図 13. 提案手法における行列式計算の置き換え

図 13 に実行時の行列式置き換えの例を示す。この例では、変更された状態遷移行列に対し、差分情報に基づいて変更された列を除去した後、得られた小行列の部位情報をもとにハッシュテーブルを検索する。この小行列は、変更前の状態遷移行列と共通する行・列のみで構成され、ハッシュテーブルには変更前の状態遷移行列から得られるすべての小行列が部位情報の形で格納されているため、検索した部位情報と一致するレコードが必ず存在し、小行列に対応する行列式計算結果を得ることができる。

Algorithm1 に、提案手法における再帰的な余因子展開を実装するための疑似コードを示す。関数 *Cofactor-Expansion* は、再帰的に呼び出されることで行列式の置換または余因子展開を行い、行列式計算を実現する関数である。小行列の部位情報は 1 行目の *bitset* として定義される。これは取り除かれた行の集合と取り除かれた列の集合の組であり、初期状態では空集合のタプルである。3.2 節より、余因子展開によって展開される小行列は、取り除かれた行と列の集合によって表現できる。そのため、*bitset* はハッシュテーブルのキーとしてだけでなく、展開された小行列の識別情報としても利用でき、再帰

的な余因子展開によって展開される小行列の状態を追跡することができる。4行目の関数 $diff(Matrix)$ によって行列の差分情報を取得し、差分を含む行・列の集合を取り除くべき行・列の集合 $nextToExtract$ として定義する。 $bitset$ には現在取り除かれている行・列の集合が格納されているため、 $nextToExtract$ と比較することで余因子展開が必要かどうかを判断する。余因子展開が必要な場合、5行目から20行目で展開処理を行う。for文は展開された各項を表し、それぞれの項で行列の要素 P_{ij} と小行列の符号を返す関数 $getsigns$ の積を取り、小行列の行列式計算を求めるために $CofactorExpantion$ 関数を再帰的に呼び出す。最後に各項の線形和を $result$ として返す。余因子展開が不要な場合、部位情報から小行列の置き換えが可能な時、22行目でキャッシュ検索を行い、計算結果を返す。関数 $replace$ はキャッシュ検索と置換を行う関数であり、部位情報を引数として行列式計算結果を返す。

4 実験・評価

4.1 実験設定

本実験では、実行中の検証式再生成に必要なとなる行列式計算の計算時間について、提案手法と従来手法を比較してその有効性を評価する。また、検証式の再生成に必要なキャッシュサイズを評価するため、キャッシュを使用する既存手法である Fujimoto らの手法と比較する。

実験では以下のシナリオを想定する。まず設計時において、振る舞い変更前の状態遷移行列から状態遷移行列を生成し、手法ごとの必要に応じてキャッシュを保存する。次に実行時において、システムの振る舞い変更を反映した変更後の状態遷移行列から、新たな検証式を作成する。このシナリオを通じて、設計時の検証式生成に必要な行列式計算およびキャッシュ保存に要した時間と、実行時の検証式再生成に必要な行列式計算に要した時間を計測した。加えて、キャッシュに要したメモリ使用量を計測し、手法それぞれの計算時間とメモリ使用量を比較する。

状態遷移行列の変化については以下のように想定する。状態遷移行列における要素の追加・削除は想定せず、行列内の遷移確率の変化によって、少数の行が変化した場合を想定して実験を行う。図 14 に変化の例を示す。この例では大きさ 6 の状態遷移行列において、行列内の 2 行に渡って 4 要素が変化している。

Algorithm 1 再帰的な余因子展開のアルゴリズム

```

1:  $bitset$  :          ▷ 展開した各項の部位情報を追跡する
2: function COFACTOREXPANTION( $bitSet$ )
3:    $nextToExtract \leftarrow diff(Matrix)$  ▷ 差分情報を取得
4:   if 取り除くべき行が残っている then
5:      $result \leftarrow 0$                                 ▷ 返り値
6:     展開する行  $i \leftarrow nextToExtract$ 
7:     for 残った各列  $j$  do
8:        $bitset' \leftarrow$  展開部位から  $bitset$  を更新
9:        $\_tmp \leftarrow$  CofactorExpantion( $bitSet'$ )
10:       $result += \_tmp \times P_{ij} \times getsign(bitset', i, j)$ 
11:    end for
12:    return  $result$ 
13:  else if 取り除くべき列が残っている then
14:     $result \leftarrow 0$                                 ▷ 返り値
15:    展開する列  $j \leftarrow nextToExtract$ 
16:    for 残った各行  $i$  do
17:       $bitset' \leftarrow$  展開部位から  $bitset$  を更新
18:       $\_tmp \leftarrow$  CofactorExpantion( $bitSet'$ )
19:       $result += \_tmp \times P_{ij} \times getsign(bitset', i, j)$ 
20:    end for
21:    return  $result$ 
22:  else
23:    return  $replace(bitset)$           ▷ 部位情報による
    キャッシュ検索結果を返り値として返す
24:  end if
25: end function

```

実験では 3 種類の状態遷移行列を用意し、それぞれの行列が変化するパターンに対して実験を行った。用意した行列と変化のパターンは以下の通り。

1. 大きさ 10, 変数の数は 7, 吸収状態は 2 つで、行列内の 1 行に渡り 2 要素が変化する
2. 大きさ 17, 変数の数は 3, 吸収状態は 2 つで、行列内の 2 行に渡り 6 要素が変化する
3. 大きさ 25, 変数の数は 5, 吸収状態は 2 つで、行列内の 2 行に渡り 6 要素が変化する

システムモデルにおける状態数の例として、Fang らの研究 [12] では、外国為替 (FX) 市場での取引を行うシステムをモデル化し、29 状態を持つマルコフモデルを形成している。比較対象として、Fileri らの手法 [1] と

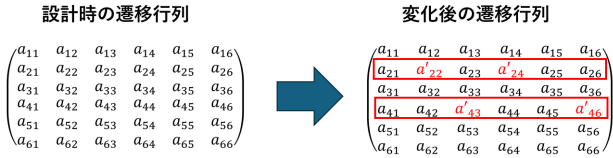


図 14. 実験での状態遷移行列変化の例

表 1. モデルごとの設計時計算時間 [ms] の比較実験結果

モデルの大きさ	Filieri らの手法	Fujimoto 手法	提案手法
10	1.0	25.0	183.1
17	5.0	139.1	989.3
25	638.6	15699.6	284235.9

Fujimoto らの提案手法 [5] を用意し、提案手法を含む 3 手法と設計時の計算時間と実行時の計算時間を比較した。また、Fujimoto らの手法とメモリ使用量の比較を行った。実行時間の単位は [ms] であり、メモリ使用量の単位は [MB] である。

4.2 実験結果

4.2.1 実行時間比較

表 1 および表 2 では用意した 3 種類のモデルそれぞれにおける計算時間の実験結果を示している。設計時の実行時間を比較した表 1 が示すように、すべてのモデルに共通して提案手法が最も長く、キャッシュを使用する Fujimoto 手法が次に長くなった。実行時の計算時間について比較した表 2 が示すように、提案手法は既存の 2 手法と比較して、大きさ 10 および 17 のモデルに対しては計算時間がより長くなっている。一方で既存の 2 手法は行列の大きさ増加に伴い実行時の計算時間が指数関数的に増大するのに対し、提案手法の実行時間増加は比較的緩やかであり、大きさ 25 のモデルに対してはもっとも実行時間が短くなった。

4.2.2 メモリ使用量比較

表 3 では 3 種類のモデルそれぞれにおいて、提案手法と Fujimoto 手法でキャッシュの保存に必要としたメモリ使用量の実験結果を示している。両手法において、サイズが大きくなるほどメモリ使用量が増大している。手法ごとに比較すると、サイズ 10 および 17 の場合は提案手

表 2. モデルごとの実行時計算時間 [ms] の比較実験結果

モデルの大きさ	Filieri らの手法	Fujimoto 手法	提案手法
10	1.4	0.9	5.0
17	5.5	11.2	6.8
25	599.6	506.1	11.4

表 3. モデルごとの使用メモリ量 [MB] の比較実験結果

モデルの大きさ	Fujimoto 手法	提案手法
10	0.012	0.058
17	0.303	0.735
25	19.309	4.167

法の方がメモリ使用量が比較的大きく、サイズ 25 の場合は提案手法の方が比較的小さくなった。

5 考察

実行時間に関する比較実験では、大きさが 25 の場合に限り、提案手法がより高速に検証式を生成することが確認できた。Filieri らの手法において状態遷移行列が変化した場合、検証式を再生成する際に設計時と同じ処理を行うため、実行時の計算時間は設計時の計算時間と同等となる。そのため大きなサイズのモデルに対する実験では、行列サイズの増大に伴う行列式計算コストの指数関数的増加により、実行時の計算時間が急増した。Fujimoto 手法では、実行時の行列式計算を部分的に削減することで実行時の計算時間を抑えることを目指したが、行列のサイズ増大に伴う実行時間の急増を解決するには至らなかった。一方、提案手法では大きさの増加に伴う実行時の計算時間増大は比較的緩やかであることが確認された。また、より大きいサイズの状態遷移行列に対しても実行時の計算時間を削減できると考えられ、提案手法は大きいサイズのモデルに対する動的検証において有効であることが示唆された。

キャッシュサイズに関する比較実験では、大きさが 10 の場合と 17 の場合では Fujimoto 手法のキャッシュサイズが比較的良かったが、大きさが 25 の場合に逆転し、提案手法のキャッシュサイズがより低い結果となった。考えられる理由としては以下のものが考えられる。Fujimoto 手法および提案手法では設計時における計算時間を抑制するため、保存する候補行列や小行列の探索を打ち切る処理を行う。そのため、両手法での打ち切り処理の違い

により、提案手法がより早期に打ち切られた結果、キャッシュサイズの逆転が発生したものと考えられる。

また、本手法は起こり得るあらゆる行列式計算を設計時におこなう手法であるため、行列のサイズ増大による設計時間への影響を最も受けやすい。その結果、提案手法はすべての場合で設計時の実行時間が最も長く、より大きいサイズのモデルに対する動的検証において問題となる可能性がある。設計時の計算時間を含めた全体の計算コスト低減が課題であり、今後の改善が求められる。また、本研究では状態遷移行列における状態の増加・削減について検討を行っていないため、行列内での状態遷移確率が変化した場合以外での有用性についての検証が必要であり、今後の研究における課題となる。

6 おわりに

本研究では、自己適応システムにおける確率的モデル検査を用いた動的検証を高速化するため、先行研究で繰り返される余因子展開処理に着目し、その性質を利用した手法を提案し、性能を検証した。Filieriらの手法では、変数を含む検証式を用いることで、変化する環境に対応した動的検証を実現した。しかし大規模な環境変化によりモデルが変化し、検証式を再生成する必要が出た場合、高速な動的検証を実現できなくなる。この問題に対して本研究では、余因子展開の性質を利用し、小行列の部位情報を利用することで、実行時の差分を取り除いた小行列を確実に置き換える手法を提案した。これにより無駄な余因子展開を減らすことで検証式の再生成を高速化することを試みた。提案手法の評価のため、サイズの異なる3つのモデルで検証式再生成を想定した実験を行った。その結果、行列サイズが増加しても実行時の計算時間増加が抑制できていることが確認された。

本研究における今後の課題は大きく3つ存在する。1つ目は様々な環境変化に対する適応可能性に関する課題である。本研究では、振る舞いの変化がシステムの一部に限定され、状態の増減が起きないという仮定の下実験を行った。しかし実際の環境変化における状態遷移行列の変化では、状態の追加や削除を含む様々なパターンが考えられる。したがって状態の追加や削除に対応する必要がある。2つ目は設計時の実行時間増大という課題である。提案手法は設計時にあらゆる小行列の場合分けを列挙し、その行列式計算を行う。そのため従来手法と比較して設計時の計算時間が非常に長いと、これを削減

する手法が求められる。3つ目はキャッシュサイズが増大する課題である。キャッシュを用いる Fujimoto 手法と提案手法は、モデルサイズの増加に伴いキャッシュサイズが指数的に増加する。そのため、提案手法をより大きいモデルサイズに適用した場合、キャッシュサイズの指数的増大が自己適応化の障害になりうる。したがって、モデルサイズの増加によるキャッシュサイズ増加を抑える手法が必要となる。

参考文献

- [1] A. Filieri, C. Ghezzi, and G. Tamburrelli, “Run-time efficient probabilistic model checking,” *ICSE 2011*, p. 341–350, 2011.
- [2] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, 2009.
- [3] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, “Model evolution by run-time parameter adaptation,” in *IEEE 31st International Conference on Software Engineering*, pp. 111–121, 2009.
- [4] R. De Lemos, Giese, *et al.*, “Software engineering for self-adaptive systems: A second research roadmap,” in *International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pp. 1–32, 2013.
- [5] M. Fujimoto, H. Nakagawa, and T. Tsuchiya, “Expansion mechanism for runtime verification of self-adaptive systems,” *SEKE 2023*, pp. 76–81, 2023.
- [6] 大須賀昭彦, 田原康之, 中川博之, and 川村隆浩, “マルチエージェントによる自律ソフトウェア設計・開発,” コロナ社, 2017.
- [7] 土屋達弘, “モデル検査入門,” 計測と制御 第48巻第11号, pp. 797–802, 2009.
- [8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.
- [9] C. M. Grinstead and J. L. Snell, *Grinstead and Snell’s Introduction to Probability*. American Mathematical Society, 1997.
- [10] H. Nakagawa, K. Ogawa, and T. Tsuchiya, “Caching strategies for run-time probabilistic model checking,” in *Proc. of the 11th International Workshop on Models@Run.Time*, vol. 1742, pp. 18–25, 2016.
- [11] H. Nakagawa, H. Toyama, and T. Tsuchiya, “Expression caching for runtime verification based on parameterized probabilistic models,” *Journal of Systems and Software*, vol. 156, p. 300–311, Oct. 2019.
- [12] X. Fang, R. Calinescu, S. Gerasimou, and F. Alhwikem, “Fast parametric model checking with applications to software performability analysis,” *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4707–4730, 2023.