抽象状態機械の到達性解析による安全性の簡易検証

大矢野　　潤

1　はじめに

電子商取引など、安全なネットワーク上のビジネスの実現には、そのビジネスモデルが望ましい性質を持っていることを設計段階で検証する必要がある。ここでいうビジネスモデルとは、本質的にはネットワーク上のコミュニケーションをともなう並列プロセスであり、その検証はオペレーティングシステムやネットワーク、計算論の分野で盛んに研究されてきた。

検証手続きに必要な計算量は、並列プロセスの満たすべき性質とそのモデルの大きさによって決定される。並列プロセスの満たすべき性質には安全性 (Safety)、活性 (Liveness)、公平性 (Fairness) などがあるが、それぞれ数学的に異なる性質をもつことが知られている。ここで、活性と公平性に関する検証手続きは安全性のそれよりもはるかに計算量が大きいこと、安全性には「二重に代金を支払わないこと」や「在庫がなくならないこと」など、日常のビジネスにおいて基本的な性質を含んでいることから、本研究では特に安全性に注目する。

本論文では、抽象状態機械の到達性解析による並列プロセスの安全性検証について述べる。安全性は、「悪い状態に到達しない」という到達性を解析することで検証可能であり、また、モデルの大きさは同様な状態を同一視することで低く抑えられることが期待できる。実証実験として、排他制御アルゴリズムであるピーターソンのアルゴリズムを検証するためのシンプルなアルゴリズムと、その実装を行った。その結果、ピーターソンのアルゴリズム程度のプロセスの検証であれば、普通のノート PC であっても 0.05 秒未満で解析が終了することがわかった。

論文の構成は次のようになっている。次節では、モデル検査に関するいくつかの数学的準備を行う。特にプロセスのモデルとしてのクリプキ構造、時相論理とその意味論、モデル探査アルゴリズムを導入する。第 3 節では、本研究で検証するピーターソンのアルゴリズムを紹介する。第 4 節では、python による簡易モデル検査機の具現化を行い、最終節で結果と考察を示す。

2　モデル検査

近年、ハードウェアや、オペレーティングシステム、公開鍵基盤のためのセキュリティプロトコルの安全性の検証など、並列システムの振る舞いの理論的枠組み [4] の整備が行われている。特に、モデル検査 (Model Checking) と呼ばれる、システムのもつ性質を機械的、網羅的に調べる技術体系が目覚ましく発達してきており、SPIN[1]、nuSMV[2]、UPPAAL[3]、など有用なツールも多数、無償で提供されている。加えて、この分野の先駆者による教科書 [3, 7] も充実しており、情報系学部、大学院等での講義も盛んに行われている。モデル検査の理解には多岐にわたる知識が必要である

[1] http://spinroot.com/spin/whatispin.html
[2] http://nusmv.fbk.eu/
[3] http://www.uppaal.org/

が、ここでは本論文の理解に必要な最小限のものについて紹介する。論文中の定義、記法、例などは Emerson の解説論文 [4]、東京大学 萩谷昌己氏の講義録 [4] などを参考にした。

2.1 システムの満たすべき性質

複数のプロセスからなるプログラムの満たすべき性質として Leslie Lamport により導入された安全性と活性が代表的である。Lamport は文献 [6] において安全性と活性をそれぞれ “something will not happen”、“something must happen” としている。安全性でいう “something” とはデッドロックやプロセスの競合などの「何か悪いこと (something bad)」を意味しているのに対し、活性の “something” は CPU の割り当てを受けるなどの「何かいいこと (something good)」を指している。これらに加えて、公平性が主張されることも多い。例えば、あるプロセスに CPU の割り当てが集中し、他のプロセスの実行が無期限に延期されたり資源を確保したまま解放しないプロセスを容認するといった不公平なシステムにおいては、排他制御は常に安全に行われてしまう。このため、システム検証の前提条件として公平性が仮定されることが多い。

安全性と活性の数学的特徴づけは Bowen Alpern、 Fred B. Schneider らによって行われた [1, 2]。彼らは、安全性を閉集合、活性は稠密集合として特徴づけ、システムが満たすべきすべての性質は安全性と活性の論理積により記述できることを示し、その位相論的証明を与えた。公平性も安全性と活性の組み合わせで表現されるため、安全性と活性に関する議論が本質的に重要であることがわかる。

2.2 クリプキ構造

Alpern と Schneider はシステムを状態の無限列の集合 [2]、Büchi オートマトン [1] の受理集合として、その上の性質を定義したが、本論文では、システムをクリプキ構造により定義する。

クリプキ構造とは Saul Kripke により導入されたラベル付き状態遷移システムの一種であり、プロセスの状態を簡潔に記述できることからモデル検査においてよく使用される。

定義 **2.1** (クリプキ構造)**.** クリプキ構造は、状態の集合 S、リレーション（もしくはエッジ）の集合 R、そしてラベル付け関数 L の組 $K =< S, R, L >$ として表現される。R は $S \times S$ の部分集合であり、ラベル付け関数 L は状態から原子命題 *(Atomic Proposition)* の集合 AP の部分集合への関数 $L : S \rightarrow 2^{AP}$ である。

複数のクリプキ構造を取り扱う場合には、クリプキ構造 K の状態集合を $|K|$、エッジ集合を $K^{\rightarrow}$、ラベル付け関数を L_K、すなわち $K =< |K|, K^{\rightarrow}, L_K >$ と記述することとする。

2.3 時相論理

システムの持つ性質を表現する方法として、時間に関する語彙をもつ時相論理式を使う方法、無限の状態列を受理する ω オートマトン (Büchi、Muller、Rabin、Streett オートマトンなど) を使う方法などがある。

本研究では時間に関する様相を用いた時相論理 (Temporal Logic) を採用する。時相論理には CTL* (Computational Tree Logic *)、そのサブセットである CTL や LTL (Linear Time Logic) などが代表的である。CTL* の表現する性質は広範囲に渡るが、反面、その検証にかかる時間は現実的ではな

[4] http://hagi.is.s.u-tokyo.ac.jp/pub/staff/hagiya/kougiroku/jpf/modal-temporal.pdf

い (PSPACE-complete)。CTL と LTL は表現力においてお互いに補完する関係にあるが、CTL の計算量は比較的低く抑えられ (P-complete)、 LTL のほうがはるかに大きい (PSPACE-complete) ことが知られている。本研究の目的は安全性の検証であり、表現力を抑えた CTL の一部を用いる。論理式の表現力とモデル検査に必要な計算量に関する議論の詳細は文献 [4] を参照してほしい。

定義 **2.2** (CTL の式)**.** AP を原子命題の集合とするとき、AP 上での *CTL* 式の集合は次のように定義される。

- $a \in AP$ のとき a は *CTL* 式である
- Φ_1, Φ_2 がそれぞれ *CTL* 式のとき、次のものは *CTL* 式である

$$\neg\Phi_1, \Phi_1 \vee \Phi_2, \boldsymbol{EX}\Phi_1, \boldsymbol{EG}\Phi_1, \Phi_1 \boldsymbol{EU} \Phi_2$$

定義 **2.3** (CTL の意味)**.** *CTL* 論理式 Φ のクリプキ構造 K における意味とは、その論理式を満たす状態、およびパスの集合　$[\![\Phi]\!]_K$ で定義される。以下、s を状態、 $\pi = s_0, \ldots, s_n, \ldots$ を R と整合的 *(*すなわち、$\forall i, (s_i, s_{i+1}) \in R$*)* な状態列、$\pi(i)$ を状態列 π の i 番目の状態 s_i とする。この時、論理式の意味は次のように再帰的に定義される。

$$\begin{aligned}
[\![\texttt{True}]\!]_K &= S \\
[\![a]\!]_K &= \{s \in S \mid a \in AP \text{ and } a \in L(s)\} \\
[\![\neg\Phi_1]\!]_K &= S - [\![\Phi_1]\!]_K \\
[\![\Phi_1 \vee \Phi_2]\!]_K &= [\![\Phi_1]\!]_K \cup [\![\Phi_2]\!]_K \\
[\![\mathbf{EX}\Phi_1]\!]_K &= \{s \mid \exists t \text{ s.t. } (s,t) \in R \text{ and } t \in [\![\Phi_1]\!]_K\} \\
[\![\mathbf{EG}\Phi_1]\!]_K &= \{s \mid \exists \pi \text{ s.t. } \pi(0) = s \text{ and } \pi(i) \in [\![\Phi_1]\!]_K, \forall i \geq 0\} \\
[\![\Phi_1 \mathbf{EU} \Phi_2]\!]_K &= \{s \mid \exists \pi \text{ s.t. } \pi(0) = s \text{ and } k \geq 0 \text{ s.t. } \pi(i) \in [\![\Phi_1]\!]_K, \forall i < k \text{ and } \pi(k) \in [\![\Phi_2]\!]_K\}
\end{aligned}$$

これらを組み合わせて次のような論理式を作ることができる。

$$\begin{aligned}
&\texttt{False} \equiv \neg\texttt{True}, \qquad && \Phi_1 \wedge \Phi_2 \equiv \neg(\neg\Phi_1 \vee \neg\Phi_2) \\
&\mathbf{AX}\Phi \equiv \neg\mathbf{EX}\neg\Phi, && \mathbf{AG}\Phi \equiv \neg\mathbf{EF}\neg\Phi \\
&\mathbf{AF}\Phi \equiv \neg\mathbf{EG}\neg\Phi, && \mathbf{EF}\Phi \equiv \texttt{True}\ \ \mathbf{EU}\Phi
\end{aligned}$$

ここで、**A**、**E**、**X**、**G**、**F**、**U** オペレータはそれぞれ “All”、“Exists”、 “Next”、“Globally”、“Finally”、“Until” を表現しており、次のような解釈を与えることができる。

- **AG**Φ : Φ はすべてのパス上で常に成り立っている
- **AF**Φ : Φ がすべてのパス上でいつか成立する
- **EF**Φ : Φ がいつか成立するパスが存在する
- **EG**Φ : Φ が常に成立するパスが存在する

定義 **2.4** (単調性)**.** $\tau : 2^S \to 2^S$ が与えられた時、τ が単調であるとは $P \subseteq Q$ ならば $\tau(P) \subseteq \tau(Q)$ が成り立っていることをいう。

事実 **2.1.** *CTL* 式の解釈 $[\![-]\!]$ は単調である。

定理 **2.1** (Tarski-Knaster)**.** $\tau : 2^S \to 2^S$ が単調であるとき、

(1) $\mu Y.\tau(Y) = \cap\{Y : \tau(Y) = Y\} = \cap\{Y : \tau(Y) \subseteq Y\}$

(2) $\nu Y.\tau(Y) = \cup\{Y : \tau(Y) = Y\} = \cup\{Y : \tau(Y) \subseteq Y\}$

(3) $\mu Y.\tau(Y) = \cup_i \tau^i(\mathtt{False})$

(4) $\nu Y.\tau(Y) = \cap_i \tau^i(\mathtt{True})$

が成り立つ。ここで、$\mu Y.\tau(Y)$ は、$Y = \tau(Y)$ を満たす不動点のうち最小のもの、$\nu Y.\tau(Y)$ は最大の不動点を示している。

最小および最大不動点を用いた時相論理（命題 μ 計算）は Dana Scott、Jaco de Bakker らにより導入され、Doxer Kozen により整備された [5]。命題 μ 計算により、前出の CTL オペレータは次のように定義することができる。

$$\mathbf{AG}\Phi \equiv \nu Z.\Phi \wedge \mathbf{AX}Z, \qquad \mathbf{AF}\Phi \equiv \mu Z.\Phi \vee \mathbf{AX}Z$$
$$\mathbf{EF}\Phi \equiv \mu Z.\Phi \vee \mathbf{EX}Z, \qquad \mathbf{EG}\Phi \equiv \nu Z.\Phi \wedge \mathbf{EX}Z$$

特に定理 2.1 (3) は状態を求めるアルゴリズムを具体的に与えていることに注意したい。$\mathbf{EF}\Phi$ は、$\tau(Z) \equiv \Phi \vee \mathbf{EX}Z$ において $\mathbf{EF}\Phi \equiv \mu Z.\tau(Z)$ と定義される。τ は単調であることから次の近似上昇列が得られ、

$$\mathtt{False} = \tau^0(\mathtt{False}) \subseteq \tau(\mathtt{False}) \subseteq \tau^2(\mathtt{False}) \subseteq \ldots \subseteq \tau^k(\mathtt{False}) = \tau^{k+1}(\mathtt{False})$$

となる [4]。すなわち、モデル K の状態集合 S の濃度 $\#S$ を超えない最小の $0 \leq k \leq \#S$ が存在し、$\tau^k(\mathtt{False}) = \tau^{k+1}(\mathtt{False})$ となることが定理 2.1 (3) により保証されている。

系 **2.1.** $\mu Z.\tau(Z)$ 任意の *CTL* 論理式 τ について、モデル K に対し、最小の $0 \leq k \leq \#S$ が存在し、

$$\mu Z.\tau(Z) = \tau^k(\mathtt{False})$$

となる。

系 2.1 より、$[\![\mathtt{False}]\!]_K (= [\![\neg\mathtt{True}]\!]_K = S - [\![\mathtt{True}]\!]_K = \emptyset)$ から始まる状態集合の上昇列の最大元 $[\![\tau^k(\mathtt{False})]\!]_K$ がモデル K における $\mathbf{EF}\Phi$ の意味となる。

2.4 到達性解析

モデル検査機が システム K について $\mathbf{AG}\neg\Phi$ が妥当である ($K \models \mathbf{AG}\neg\Phi$) と告げた場合は、そのシステムでは不都合が起きない、つまり安全であることの証明となる。ここで、Φ をデッドロックを表す命題であるとすると、論理式 $\mathbf{AG}\neg\Phi$ は “すべての選択肢において常にデッドロックが起こらない” ことを主張している。逆に、モデル検査が失敗した場合には、$K, \pi \not\models \mathbf{AG}\neg\Phi$ となる反例 π をシステム設計者に告げる。CTL において

$$\begin{aligned} K, \pi \not\models \mathbf{AG}\neg\Phi &\Leftrightarrow K, \pi \models \neg\mathbf{AG}\neg\Phi \\ &\Leftrightarrow K, \pi \models \neg(\nu Z.\mathbf{AX}Z \wedge \neg\Phi) \\ &\Leftrightarrow K, \pi \models \mu Z.\mathbf{EX}Z \vee \Phi \\ &\Leftrightarrow K, \pi \models \mathbf{EF}\Phi \end{aligned}$$

であるため、モデル検査機が返した反例 π は、いつか Φ が成り立つパス π が存在していることの証拠となる。システム設計者はこの証拠 π を解析することにより、システムがデッドロックの状態に到達するまでの具体的な経路を求めることができる。経路を求める具体的なアルゴリズム **1** は、定理 2.1 (3) より直接導出することができる。

アリゴリズム **1** $[\![\mathbf{EF}\phi]\!]$

```
X ← ∅
loop
    Y ← [[φ]] ∪ {s ∈ S | ∃t ∈ S s.t. R(s,t) ∧ t ∈ X}
    if Y = X then
        break
    end if
    X ← Y
end loop
```

2.5 抽象状態機械

定義 2.5 (抽象モデル)**.** クリプキ構造 K、K' について、すべての K の状態 s に対し K' の状態 s' が存在し、s の命題集合と s' の命題集合との間に包含関係 $L(s) \subseteq L'(s')$ が成立するものとする。さらに、すべての K のエッジ (s,t) に対し K' のエッジ (s',t') が存在するとき、K' は K の抽象モデル、逆に K は K' の具体モデルであるといい、$K \preceq K'$ と記述する。

$K \preceq K'$ のとき、K のパスに対応するパスが K' で存在しているということは、**AG**¬Φ、すなわち、“すべてのパス上で常に ¬Φ が成り立つ” という主張が K' 上で成立するのであれば、同じ性質はモデル K 上でも成立していることを意味している。すなわち、$K' \models$ **AG**¬Φ の結果が肯定的であれば、その結果を用いて $K \models$ **AG**¬Φ と結論してよい。

逆に、 モデル検査機が否定的な応答をした場合、つまり $K', \pi \not\models$ **AG**¬Φ という判定結果から $K, \pi \not\models$ **AG**¬Φ を結論することはできない。K' で見つかったパスが K に存在するとは限らないからである。もし、K' の反例 π が K に存在しない場合には π をみせかけ (spurious) の反例であるという。

図 1 では、具体モデルの $c0$ が抽象モデルの $a0$ に、$c1$、$c2$ が $a1$ に、$c3$ が $a2$ に対応しており、エッジ $c0 \to c1$、$c2 \to c3$ はそれぞれ $a0 \to a1$、$a1 \to a2$ に対応していることを示している。この時、抽象グラフではノード $a0$ から $a2$ へのパス $a0 \to a1 \to a2$ が存在しているが、具体グラフ上で $a0$ に対応するノード $c0$ から $a2$ に対応するノード $c3$ へのパス $c0 \to \ldots \to c3$ は存在しない。すなわち、$a0 \to a1 \to a2$ はみせかけのパスである。

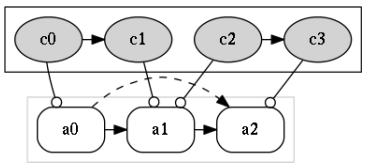

図 1: みせかけのパス

3 ピーターソンのアルゴリズム

ピーターソンのアルゴリズムとは、2 つのプロセス (以下、“プロセス me” と “プロセス you” とする) 間での排他制御をおこなうための、通信用の共有メモリを使用したアルゴリズムである。

共有メモリには、フラグ変数 *flags*[me]、*flags*[you] と 変数 *turn* があり、*flags*[me] の値が True のとき、プロセス me がクリティカルセクションに入りたいという意思があることを、同様に *flags*[you] はプロセス you がクリティカルセクションに入ろうとしていることを示している。変数 *turn* は、その時点で優先権を持つプロセスを示している。すなわち、プロセス me がクリティカルセクションに入るには、そもそもプロセス you がクリティカルセクションに入ろうとしていないか、競合する場合には、変数 *turn* によりプロセス me に優先権が与えられているときに限る。加えて、プロセスはクリティカルセクションから出るときに *turn* の値を他のプロセスに設定することで他のプロセスに優先権を与えることはできるが、自身に設定することはできない。

この仕組みにより、

- 複数のプロセスが競合する場合、*turn* で設定されているプロセスのみクリティカルセクションに入ることができる（排他制御）
- 他のプロセスのクリティカルセクションに入ろうとしている時に、連続してクリティカルセクションに入ることはできない（飢餓状態の回避）

を実現している。

アリゴリズム **2** ピーターソンのアルゴリズム

```
// プロセス me                          // プロセス you
loop                                    loop
    flags[me] ← True                        flags[you] ← True
    turn ← you                              turn ← me
    while flags[you] =True do               while flags[me] =True do
        if turn ≠ you then                      if turn ≠ me then
            break                                   break
        end if                                  end if
    end while                               end while
    クリティカルセクション                  クリティカルセクション
    flags[me] ← False                       flags[you] ← False
    アイドリング                            アイドリング
end loop                                end loop
```

アルゴリズム **2** をプログラムカウンタを用いて等価な表現にしたものがアルゴリズム **3** であり、プログラムカウンタと状態とみなし、状態遷移図に変換したものが図 2 である。それぞれの図において、四角形で囲まれた状態 0 がプロセスの初期状態を、八角形で囲まれた状態 4 はクリティカルセクションに入ったことを示している。また、二重丸で囲まれた状態 6 は、セルフループがあることを示している。

4 具現化

ここでは、ピーターソンアルゴリズムから抽象モデルを構築し、そのモデル上でクリティカルセクションに到達する可能性のある状態の集合を求める手続きについて説明する。本手続きは、大ま

アリゴリズム **3** ピーターソンのアルゴリズム (プログラムカウンタ版)

// プロセス me	// プロセス you
0: *flags*[me] ← True	0: *flags*[you] ← True
1: *turn* ← you	1: *turn* ← me
2: **if** *flags*[you] ≠True **then goto** 4	2: **if** *flags*[me] ≠True **then goto** 4
3: **if** *turn* ≠ you **then goto** 4 **else goto** 2	3: **if** *turn* ≠ me **then goto** 4 **else goto** 2
4: クリティカルセクション	4: クリティカルセクション
5: *flags*[me] ← False	5: *flags*[you] ← False
6: **ether goto** 6 **or goto** 0	6: **ether goto** 6 **or goto** 0

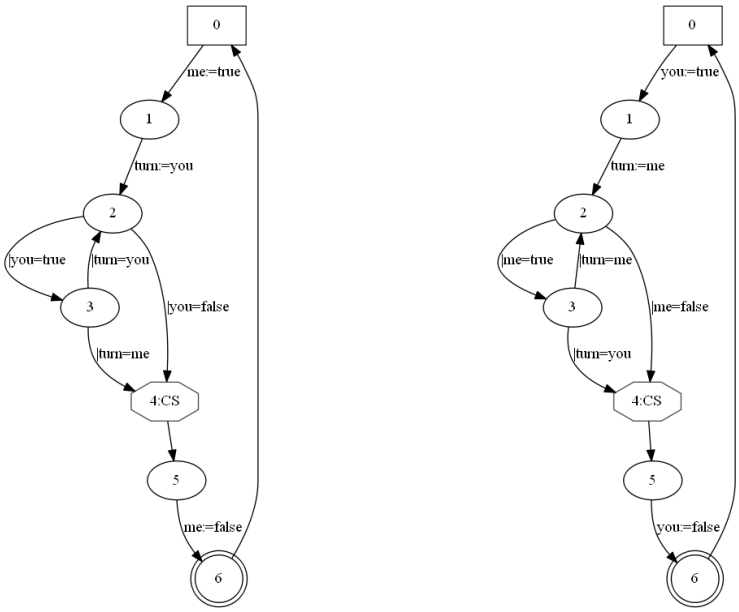

図 2: ピーターソンのアルゴリズムの状態遷移

かに次のステップからなる。

1. ピーターソンアルゴリズムの状態遷移定義ファイルを読み込み、プロセス me、プロセス you それぞれに対応する状態遷移グラフを構築する
2. 命題 Φ をクリティカルセクションとし、Φ に到達する可能性のある状態遷移、すなわち **EF**Φ に対応する抽象グラフを構築する
3. 開始ノードから Φ に対応する状態へのパスを含むサブグラフを構築する
   - サブグラフが空の時、開始ノードからクリティカルセクションへの計算は存在しない
   - それ以外の時、対応する具体グラフにおいて可達かどうかを判定する

4.1 プロセス定義ファイル

プロセス定義は YAML(Ain't Markup Language) [5] を用いた。YAML は XML (eXtensible Markup Language) よりも人間にとって可読性の高いデータ記述言語であり、さまざまなプログラミング言語から利用可能である。ここでは、クリプキ構造 $K =< S, R, L >$ を定義するため、状態の集合 S、エッジ（もしくはリレーション）の集合 R を定義している。通常、ラベル L は状態から命題の集合 AP の部分集合への関数 $L : S \rightarrow 2^{AP}$ として定義されるが、ここでは LS として定義している。加えて、状態遷移にともなう状態の変化を規定するために、エッジにもラベル付け関数 LR を定義している。LR は定義ファイルからクリプキ構造をつくる時点で使用される。

例として、プロセス me の YAML ファイルを図 3 に示す。各項目の具体的な意味は図中にコメント文 (#コメント) として示している。

```
S: [0,1,2,3,4,5,6]                  #状態の集合
R:                                  #それぞれの状態から到達可能なノードの集合
   0 : [1]                          # 0 -> 1
   1 : [2]                          # 1 -> 2
   2 : [3,4]                        # 2 -> 3 and 2 -> 4
   3 : [2,4]                        # 3 -> 2 and 3 -> 4
   4 : [5]                          # 4 -> 5
   5 : [6]                          # 5 -> 6
   6 : [6, 0]                       # 6 -> 6 and 6 -> 0
LS:                                 # LS: 状態 -> 2^AP
   4 : [CS]                         # クリティカルセクション
LR:                                 # エッジのラベル
   0:
      1: "me:=true"                 # 0 -"me:=true"  -> 1
   1:
      2: "turn:=you"                # 1 -"turn:=you" -> 2
   2:
      3: "|you=true"                # 2 -"|you=true" -> 3
      4: "|you=false"               # 2 -"|you=false"-> 4
   3:
      2: "|turn=you"                # 3 -"|trun=you" -> 2
      4: "|turn=me"                 # 3 -"|trun=me"  -> 4
   5:
      6: "me:=false"                # 5 -"me:=false" -> 6
```

図 3: プロセス me の定義

4.2 抽象グラフの構築

次に、"プロセス me" と "プロセス you" のプロセス記述から得られたクリプキ構造を合成する手続きについて説明する。今後、変数 x の次のステップでの値を x' と記述する。

図 2 で定義された状態遷移に、そのプロセスが操作/参照する変数を明示的に記述したものが具体グラフのノード (状態) となる。

$$\{(s_{\mathtt{me}}, s_{\mathtt{you}}, me, you, turn) \mid s_{\mathtt{me}}, s_{\mathtt{you}} \in \{0, 1, \ldots, 6\}, me, you \in \{\mathtt{True}, \mathtt{False}\}, turn \in \{\mathtt{me}, \mathtt{you}\}\}$$

[5] http://yaml.org/

具体グラフは、取りうるすべての値を状態として持つため、可達でない無駄なノードを大量に生成してしまう。これを避けるため、次のように抽象化したモデル $K_{\mathtt{MTU}}$ を構築する。

$$|K_{\mathtt{MTU}}| = \{(s_{\mathtt{me}}, s_{\mathtt{you}}, me, you, turn) |\ s_{\mathtt{me}}, s_{\mathtt{you}} \in \{0,1,\ldots,6\}, me, you \subseteq \{\mathtt{True}, \mathtt{False}\}, turn \subseteq \{\mathtt{me}, \mathtt{you}\}\}$$

すなわち、変数の状態として値の集合をとる。しかし、すべての値のべき集合を構築するとかえって状態数は爆発してしまうため、ある状態 s' が与えられた時、その状態に遷移可能な状態 s のうち、最も一般的な状態 1 つを実際に生成する。この判断をしているのが関数 setPrevState である。setPrevState は表 1 のように、遷移が妥当かどうか (*linkValid*) を判定するための前/後条件、現在の状態 (*curNode*) に対応する入力状態 (*prevNode*) を規定している。

ラベル	前条件	後条件	入力状態 (出力状態との差分)
you:=true	無条件	$\mathtt{True} \in you$	$you' \leftarrow \{\mathtt{True}, \mathtt{False}\}$
me:=true	無条件	$\mathtt{True} \in me$	$me' \leftarrow \{\mathtt{True}, \mathtt{False}\}$
turn:=you	無条件	$\mathtt{you} \in turn$	$turn' \leftarrow \{\mathtt{me}, \mathtt{you}\}$
turn:=me	無条件	$\mathtt{me} \in me$	$turn' \leftarrow \{\mathtt{me}, \mathtt{you}\}$
|you=true	$you' = \{\mathtt{True}\}$	$\mathtt{True} \in you$	$you' \leftarrow \{\mathtt{True}\}$
|you=false	$you' = \{\mathtt{False}\}$	$\mathtt{False} \in you$	$you' \leftarrow \{\mathtt{False}\}$
|me=true	$me' = \{\mathtt{True}\}$	$\mathtt{True} \in me$	$me' \leftarrow \{\mathtt{True}\}$
|me=false	$me' = \{\mathtt{False}\}$	$\mathtt{False} \in me$	$me' \leftarrow \{\mathtt{False}\}$
|turn=you	$turn' = \{\mathtt{you}\}$	$\mathtt{you} \in turn$	$turn' \leftarrow \{\mathtt{you}\}$
|turn=me	$turn' = \{\mathtt{me}\}$	$\mathtt{me} \in turn$	$turn' \leftarrow \{\mathtt{me}\}$
(空ラベル)	無条件	無条件	なし

表 1: setPrevState

抽象モデル構築アルゴリズムはアルゴリズム **4** に示したとおり、あるクリティカルセクションを示す状態のうち、もっとも一般的な状態 (この場合は、$(4, 4, \{\mathtt{True}, \mathtt{False}\}, \{\mathtt{True}, \mathtt{False}\}, \{\mathtt{me}, \mathtt{you}\})$) に到達可能な状態をプロセス me、プロセス you のそれぞれの状態遷移のエッジを逆に辿ることにより探査していく。例えば、遷移 $s \to s'$ のラベルが "|me=true" であるにも関わらず、状態 s' の me 変数の値が False を示している場合には、対応する入力状態を構築し、その間のエッジが非連結である印 ("disconnected") をつけ、その後の探査を打ち切る。

続いて、手続き makeMTU に関するいくつかの性質を示す。

補題 4.1. makeMTU を有限モデルの任意のペア K_M, K_U に対して適用した場合、常に停止する。

証明. makeMTU は与えられたモデルのペア M, U のエッジ $M^{\rightarrow}$、$U^{\rightarrow}$ を合成して作ったエッジ $\{((s,t),(s',t)) \mid (s,s') \in K_M^{\rightarrow}, t \in |K_U|\} \cup \{((s,t),(s,t')) \mid (t,t') \in K_U^{\rightarrow}, s \in |K_M|\}$ に対し高々一度実行される。有限モデルのペア K_M, K_U のエッジから合成されたエッジも有限であるため、makeMTU は有限回の実行で終了する。 □

次に、モデル K と論理式 Φ に対し、K 上で Φ を満足させる状態の集合とその証拠を保持するエッジの集合に制限したモデル K_Φ を定義する。なお、ここでは Φ として

$$\Phi = a \in AP \mid Z \mid \Phi_1 \vee \Phi_2 \mid \mathbf{EX}Z$$

と、CTL 式を制限した形を用いる。

アリゴリズム **4** ピーターソンのアルゴリズムの状態遷移グラフ構築

procedure MAKEMTU($s_{\mathtt{me}}, s_{\mathtt{you}}, me, you, turn$) // $K_{\mathtt{MTU}} \leftarrow K_{\mathtt{me}} \times K_{\mathtt{you}}$
  $curNode \leftarrow (s_{\mathtt{me}}, s_{\mathtt{you}}, me, you, turn)$
  for $edge \in$INEDGES($s_{\mathtt{me}}$) **do**
    $label \leftarrow$GETLABEL($edge$)
    $prevNode, linkValid \leftarrow$ SETPREVSTATE($s_{\mathtt{me}}$, source node of $edge, label$)
    if $prevNode \notin |MTU|$ **then**
      ADDNODE($prevNode$)
    end if
    if $linkValid$ **and** $(prevNode, curNode) \notin K^{\rightarrow}_{\mathtt{MTU}}$ **then**
      ADDEDGE($prevNode, curNode$)
      $(s'_{\mathtt{me}}, s'_{\mathtt{you}}, me', you', turn') \leftarrow prevNode$
      MAKEMTU($s'_{\mathtt{me}}, s'_{\mathtt{you}}, me', you', turn'$)
    else
      ADDEDGE($prevNode, curNode, status =''$ disconnect$''$)
    end if
  end for
  for $edge \in$INEDGES($s_{\mathtt{you}}$) **do**
    プロセス you の状態についても同様の探査
    ...
  end for
end procedure

$K_{\mathtt{me}} \leftarrow$ プロセス me に対応する状態遷移グラフ
$K_{\mathtt{you}} \leftarrow$ プロセス you に対応する状態遷移グラフ
$K_{\mathtt{MTU}} \leftarrow \emptyset$
MAKEMTU(4, 4, {True,False}, {True,False}, {me,you})

定義 **4.1** (K_Φ)**.** モデル K を Φ に制限したモデル K_Φ を次のように定義する。

$$\begin{aligned} K_{a\in AP} &= < \{s \mid a \in L_K(s)\}, \emptyset, L_K > \\ K_{\Phi_1 \vee \Phi_2} &= < |K_{\Phi_1}| \cup |K_{\Phi_2}|, K^{\rightarrow}_{\Phi_1} \cup K^{\rightarrow}_{\Phi_2}, L_K > \\ K_{\boldsymbol{EX}\Phi} &= < \{s \mid (s,t) \in K^{\rightarrow} \ s.t.\ t \in K_\Phi\}, \{(s,t) \mid (s,t) \in K^{\rightarrow} \ s.t.\ t \in K_\Phi\}, L_K > \end{aligned}$$

ここで、$\tau(Z) = \Phi \vee \mathbf{EX}Z$ とするとき、系 2.1 より、それぞれのモデル K に対し最小の $k \leq \#|K|$ が存在し

$$K_{\mathbf{EF}\Phi} = K_{\mu Z.\tau(Z)} \equiv K_{\tau^k(\mathtt{False})}$$

とできる。次のことは明らかであろう。

補題 **4.2.** $K_{\mu Z.\Phi \vee \boldsymbol{EXZ}}, s \models \mu Z.\Phi \vee \boldsymbol{EXZ}$ *iff* $K, s \models \mu Z.\Phi \vee \boldsymbol{EXZ}$

定理 **4.1.** 状態 $s \in |K|$ 対して SETPREVSTATE が $prevNode$ として返す状態 $\alpha(s)$ と、対応するエッジを持つモデルを K^α とする。加えて、MAKEMTU が生成する状態を $K^\alpha_{\text{MAKEMTU}}$ とする時、

$$K_{\mu Z.\Phi \vee \boldsymbol{EXZ}} \preceq K^\alpha_{\mu Z.\Phi \vee \boldsymbol{EXZ}} = K^\alpha_{\text{MAKEMTU}}$$

となる。

証明**.** CTL 論理式 Φ の構造に関する帰納法により証明する。

$\Phi = a \in AP$ の時、$a \in L_K(s)$ となる s に対し、$s \in \alpha(s)$、$L_K(s) \subseteq L_{K^\alpha}(\alpha(s))$ が成り立つため、$K_a \preceq K^\alpha_a$ である。

同様に、$K_{\Phi_1} \preceq K^{\alpha}_{\Phi_1}$、$K_{\Phi_2} \preceq K^{\alpha}_{\Phi_2}$、の時 K_{Φ_1}, K_{Φ_2} に対応する状態とエッジが $K^{\alpha}_{\Phi_1}$、$K^{\alpha}_{\Phi_2}$ に存在しているため、$K_{\Phi_1 \vee \Phi_2} = K_{\Phi_1} \cup K_{\Phi_2} \preceq K^{\alpha}_{\Phi_1} \cup K^{\alpha}_{\Phi_2} = K^{\alpha}_{\Phi_1 \vee \Phi_2}$ となる。

次に、$K_{\Phi} \preceq K^{\alpha}_{\Phi}$ のとき、$K_{\mathbf{EX}\Phi} \preceq K^{\alpha}_{\mathbf{EX}\Phi}$ を示す。任意の $s,t \in |K|$ where $(s,t) \in K^{\rightarrow}, t \in |K_{\Phi}|$ に対し、エッジが (s,t) がモデル K において正しい遷移であれば setPrevState はそれぞれの状態に $\alpha(s), \alpha(t) \in K^{\alpha}_{\mathbf{EX}\Phi}$ を、エッジに $(\alpha(s), \alpha(t)) \in K^{\alpha\rightarrow}_{\mathbf{EX}\Phi}$ を対応させる。すなわち、$K_{\mathbf{EX}\Phi} \preceq K^{\alpha}_{\mathbf{EX}\Phi}$ である。

$\tau(Z) = \Phi \vee \mathbf{EX}Z$ の時、帰納法の仮定により $K_{\tau^i(\Phi)} \preceq K^{\alpha}_{\tau^i(\Phi)} \Rightarrow K_{\tau^{i+1}(\Phi)} \preceq K^{\alpha}_{\tau^{i+1}(\Phi)}$ である。

モデル K に対して $\tau^k(\mathtt{False}) = \tau^{k+1}(\mathtt{False})$ となる最小の k が決まり、$K_{\mu Z.\tau(\mathtt{False})} = K_{\tau^k(\mathtt{False})} \preceq K^{\alpha}_{\tau^k(\mathtt{False})}$ となる。

最後に、$K^{\alpha}_{\tau^k(\mathtt{False})}$ のつくり方は、makeMTU のアルゴリズムと一致するため、$K_{\mu Z.\tau(\mathtt{False})} \preceq K^{\alpha}_{\text{makeMTU}}$ が言える。 □

系として次を得る。

系 **4.1.**

$$\alpha(s) \notin |K_{\text{makeMTU}}| \Rightarrow K_{\boldsymbol{EF}\Phi}, s \not\models \mu Z.\Phi \vee \boldsymbol{EX}Z$$

4.3 実行結果

makeMTU をピーターソンのアルゴリズムのプロセス you、プロセス me、に適用して得られたモデル K_{MTU} の開始ノード、すなわち $(s_{\mathtt{me}}, s_{\mathtt{you}}) = (0,0)$ をもつ状態からクリティカルセクションを示す状態 $(s_{\mathtt{me}}, s_{\mathtt{you}}) = (4,4)$ への遷移にエッジを制限したグラフを図 4 に示した。ここで、$(s_{\mathtt{me}}, s_{\mathtt{you}}) = (0,0)$ から出ているエッジに対応する矢印が破線 (⇢) すなわち無効なエッジになっていることに注意してほしい。

これにより、$\alpha((0,0)) \notin K_{\text{makeMTU}}$ が、系 4.1、補題 4.2 より $K, (0,0) \not\models \mu Z.(4,4) \vee \mathbf{EX}Z$ が導かれる。すなわち、ピーターソンのアルゴリズムから生成したグラフにおいて、初期状態から複数のプロセスが同時にクリティカルセクションに入るノードに到達することはない。

4.4 開発環境

用途	名称	仕様等
開発マシン	SONY VAIO	SVT1312AJ
CPU	Intel Core i7-3517U	1.90 ～2.40 GHz
メモリ	PC3L-12800	8 Gbytes
OS	Windows 8.1	6.3.9600
開発言語	python	2.7.9
グラフ処理	networkx	1.9.1
数値処理	numpy	1.9.1
数値処理	pygraphviz	1.2
構文解析	PyYAML	3.11

表 2: 開発環境

	具体グラフ	抽象グラフ	圧縮率
ノード数	392	144	約 37 %
エッジ数	800	144	18 %

表 3: グラフのノードとエッジの大きさ

本システムを構築するために使用した環境を表 2 に、実行時に必要となった状態数を表 3 に示す。抽象グラフのノード、エッジ数は実際に生成されたノードとエッジの数である。具体グラフは、プロセス記述に対応するグラフから単純にモデルを生成したときに生成されるモデルの状態数とエッジ数である。プロセス me、プロセス you のノード数をそれぞれ $\#|K_{\mathtt{me}}|$、$\#|K_{\mathtt{you}}|$、エッ

ジ数を $\#K^{\rightarrow}_{\mathtt{me}}$、$\#K^{\rightarrow}_{\mathtt{you}}$ 、変数 *me*、*you*、*turn* の状態数を $\#me = \#you = \#turn = 2$ としたとき、全体のノード数は $\#|K_{\mathtt{me}}| \times \#|K_{\mathtt{you}}| \times \#me \times \#you \times \#turn = 7 \times 7 \times 2 \times 2 \times 2 = 392$、エッジ数は $\#K^{\rightarrow}_{\mathtt{me}} \times \#K^{\rightarrow}_{\mathtt{you}} \times \#me \times \#you \times \#turn = 10 \times 10 \times 2 \times 2 \times 2 = 800$ となる。実際に使用したノード、エッジ数は双方ともに 144 であるため、圧縮率はそれぞれ 37%、18%となった。また、可達部のみ取り出したサブグラフのノード、エッジ数は共に 26 であり、十分に取り扱い可能な大きなグラフまで縮小することができた。さらに、YAML で記述されたモデル定義を読み込み、実際のモデル空間、および可達部のみ取り出したサブグラフを構築するのにかかる時間は 0.03 秒～0.05 秒であり、通常の PC であってもストレスなく検証できた。

5 考察

本論文では、注目する並列プロセスを抽象化した有限状態機械の到達性解析による安全性の簡易検証を提案し、具現化することでその有効性を実際に確認した。特に、ピーターソンの排他制御アルゴリズムの検証であれば、一般の PC で十分な実行効率を得ることができる。しかし、ここで用いた理論はすでによく知られているものであり、また、CTL の論理式を極端に制限した形 **EFΦ** でしか示していない。さらに、ピーターソンのアルゴリズムの検証に必要な状態空間は単純に計算したとしても、そもそも十分に小さいものであり、理論的には既存研究の域を出ない。

今後は、本研究で提案した抽象化技法を、CTL のフルセットに適用し、有効性を検証する。また、暗号プロトコル、ネットワークルーティングプロトコル、ビザンチン将軍問題など本質的に可算無限の大きさを持ち、近年のネットワークビジネスなどでその安全性の保障が求められるプロトコル群へ適応していく。

参考文献

[1] Bowen Alpern, Bowen Alpera, Fred B. Schneider, and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, Vol. 2, pp. 117–126, 1986.

[2] Bowen Alpern, Fred B. Schneider, and Communicated David Gries. Defining liveness, 1985.

[3] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[4] E. Allen Emerson. Automated temporal reasoning about reactive systems. In *Logics for Concurrency*, pp. 41–101. Springer-Verlag, 1996.

[5] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, Vol. 23, , 1983.

[6] Leslie Lamport. Proving the correctness of multiprocess programs, 1977.

[7] 林晋. プログラム検証論. 情報数学講座. 共立出版, 1995.

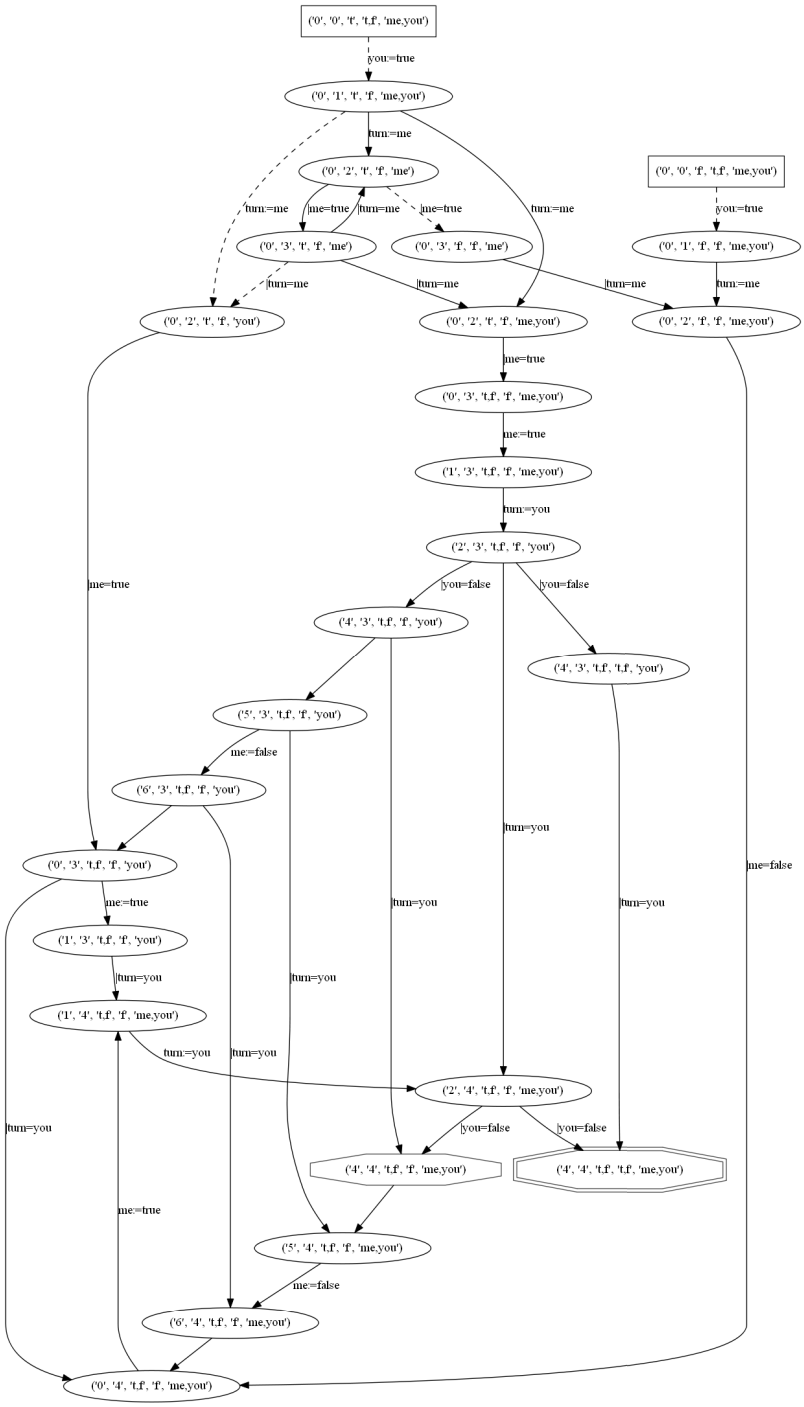

図 4: 開始ノードからクリティカル・セクションへの到達可能部分グラフ

（2015.1.22 受稿，2015.2.19 受理）

〔抄　録〕

本論文では，抽象状態機械の到達性解析による並列プロセスの安全性検証について述べる。

安全性は，「悪い状態に到達しない」という到達性を解析することで検証可能であり，また，モデルの大きさは同様な状態を同一視することで低く抑えられることが期待できる。

実証実験として，排他制御アルゴリズムであるピーターソンのアルゴリズムを検証するためのシンプルなアルゴリズムと，その実装を行った。その結果，ピーターソンのアルゴリズム程度のプロセスの検証であれば，一般に普及しているノートPCであっても0.05秒未満で解析が終了するため，十分に実用に耐えうる抽象モデルの構築とその解析環境が実現できたといえる。