修 士 論 文

状態空間探索におけるキャッシングに関する研究

指導教官　平石邦彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

小池憲史

1999 年 8 月 13 日

目 次

第 1 章

はじめに

並行プロセスは，複数の計算機上の逐次プロセスが互いに協調しながら動作を行うプロセスである．並列計算機上で動作させるプロセスや，ネットワークで接続されている計算機の為のネットワークプロトコルを包含した概念である並行プロセスについて考察する事は，これらを開発する上で重要となる．

並行プロセスでは，単一プロセスでは発生しない以下の様な問題が生じる恐れがある．

デッドロック 並行プロセスをそれ以上実行する事が出来なくなる状態のこと．

ライブロック 並行プロセスの目的の為の効果的な発展をしないまま，プロセスの実行を無限に繰り返すこと．

不適切な終了 並行プロセスの目的を達成しないまま，プロセスが終了すること．

プログラム開発中にこれらの問題を発見，解決するのは困難であり，複数のプロセスからなる並行プログラムの開発は，単一プロセスの場合と比べて難しくなっている．その為，並行プログラム作成の為の様々な手法が提案されている．

状態空間探索は並行プログラム作成支援方法の 1 つである．状態空間探索は並行プロセス中のトランジションを実行してシステム状態空間を生成し，探索する事によりプロセスに含まれる問題を発見する手法である．

状態空間探索では，一度探索した状態を再度探索するダブルワークと呼ばれる現象を抑制する為，探索した状態空間を保存しながら探索を行う．新たな状態を生成した場合，保存している状態と比べて探索済みの状態か否か判別しながら探索を行う．しかし，並行プロセスから生成される状態の数は膨大になる為，状態空間を完全に保存する事は困難である．この為，探索する状態を削減する方法が提案されている．トランジションの実行系列

内で，実行順序に依存関係の無い隣合うトランジションの交換を繰り返す事により生成された実行系列は，元の実行系列と同じ状態に到達する事を利用して探索するトランジションを削減する，スリープセットを用いた方法はその 1 つである．

また，一部の状態空間を保存するだけでダブルワークを充分に抑制出来る事が経験的に知られている．キャッシングはこの性質を利用し，キャッシュと呼ばれる高速な主記憶のみに探索した状態を保存して探索を行う手法である．主記憶量を越える状態を探索した場合には，保存されている状態から状態を選択して削除する．この為，キャッシュには再度訪問される状態を優先して保存する事が望ましい．従って，キャッシュから削除する状態を決定する方法はキャッシュを用いた探索において非常に重要な問題となる．Godefroid, Holzmann らは実験的な手法で幾つかの削除状態決定法を提案している [1][3]．しかし，これらの方法には理論的な考察が行われていない．

本研究は状態空間探索における，ダブルワークが発生するメカニズムを理論的に検証した．その結果をスリープセットを併用したキャッシングを行う際に用いるキャッシュからの削除状態決定法に適用し，Godefroid らの手法の検討を行った．さらに，それらを改良し，ダブルワークを引き起こす状態を優先してキャッシュに残す探索アルゴリズムを提案した．

本稿は次の様に構成されている．2章では，本稿で用いられている概念および用語の解説，定義を行う．3章では，キャッシングを用いた状態空間探索の際に問題となるキャッシュからの削除状態決定法について解説する．4章では，ダブルワーク発生の検証の為の並行プロセスのモデル化を行い，このモデルによるダブルワークの解析および削除状態決定法の提案を 5章で行う．最後に 6章で結論を述べる．

第 2 章

準備

本稿で用いる概念と用語の定義を以下に示す.

2.1 並行プロセス

並行プロセスは，複数の逐次プロセスが共有メモリ，メッセージ通信等を用いて協調動作を行うプロセスである．本稿では，逐次プロセス間の通信は全て共有変数によって表現されると仮定し，それらの変数をグローバル変数と呼ぶ．各逐次プロセス内でのみ有効な変数は，ローカル変数と呼ぶ.

図 2.1は並行プロセスの概念図．2 つの逐次プロセス A, B 中にそれぞれローカル変数 x, y を持ち，グローバル変数 z が存在する.

2.2 状態

逐次プロセスはプロセスに含まれる変数の内容と，現在のプロセス中の実行位置を保持している．プロセスに関するこれらの情報をシステム状態と言う．並行プロセスのシステム状態は，並行プロセスに含まれる各逐次プロセスのシステム状態およびプロセス間の共有変数の内容で構成される．本稿では，

状態 (グローバル状態) 並行プロセスのシステム状態.

ローカル状態 個々の逐次プロセスのシステム状態.

と呼ぶ．プロセスが実行されていない状態を，初期状態と呼ぶ.

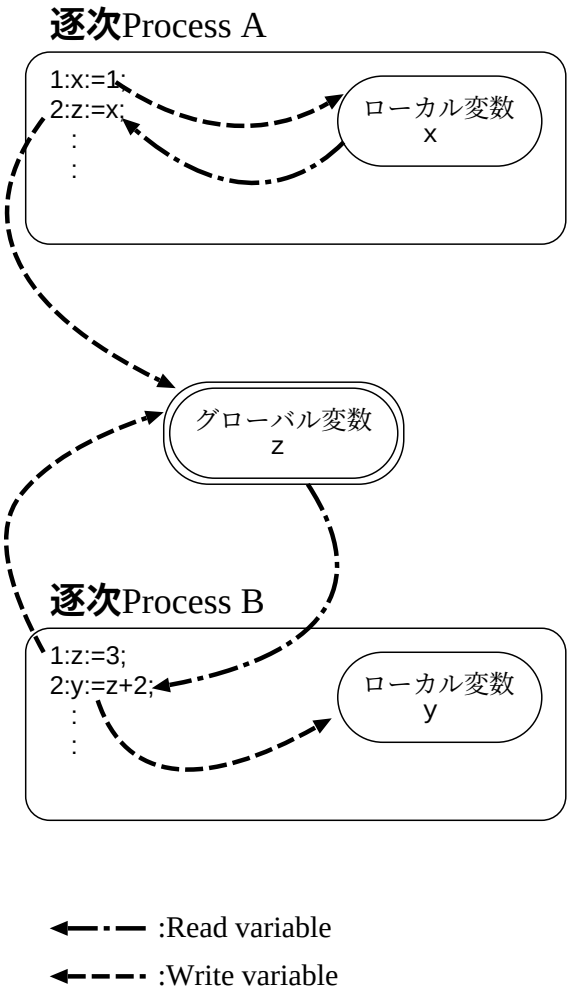

図 2.1: 並行プロセス
(:= 左辺の変数への右辺値の代入)

2.3 トランジション

トランジションは関係するプロセスにおける遷移の前後のローカル状態が等しい様なグローバルな状態遷移である．トランジションは，逐次プロセス内の命令に対応付けられた動作ラベルと，状態遷移から構成される．逐次プロセス内でのトランジションの実行順序は決定的である．プロセスの並行性により，逐次プロセス間でのトランジションの実行順序は非決定的となる．

2.4 状態空間探索

あるグローバル状態でトランジションを実行し，別の状態を生成する事を考える．トランジションは状態に含まれる幾つかのローカル状態を変化させ，新たなグローバル状態を生成する．これを並行プロセスの初期状態から繰り返し行う事により生成された状態の集合を集合を状態空間と呼ぶ．生成された状態と元となった状態の関連はトランジションにより表現される．

初期状態から実行可能なトランジションの系列を全て実行し，状態空間を生成する事を状態空間探索と呼ぶ．トランジションの実行順序の非決定性により，実行可能なトランジションが複数ある状態が存在しうる．この為，並行プロセスから生成される状態空間には膨大な数の状態が含まれる事となる．

本稿では，深さ優先でトランジション系列を実行する．実行可能なトランジションが無い状態 (終了状態) に到達した場合にはバックトラックする事により，全ての実行系列を実行する事が可能となる．なお，深さ優先探索で用いられるスタックは充分に存在するものと仮定する．

図 2.2は，図 2.1の並行プロセスの状態空間探索の様子を表している．初期状態から次々にトランジションを実行し探索を行う．トランジションが実行されると，関係する状態の要素が変化する．トランジションによる状態同士の関連付けで，状態空間を表している．

2.5 依存関係

2 つのトランジション t, t' の実行順序を入れ替えた場合，到達する状態が異なったとする．この時,「トランジション t, t' に依存関係がある」という．同じ状態に到達した場合は,「依存関係は無い」と言う．逐次プロセス内のトランジション同士には，依存関係が存在する．

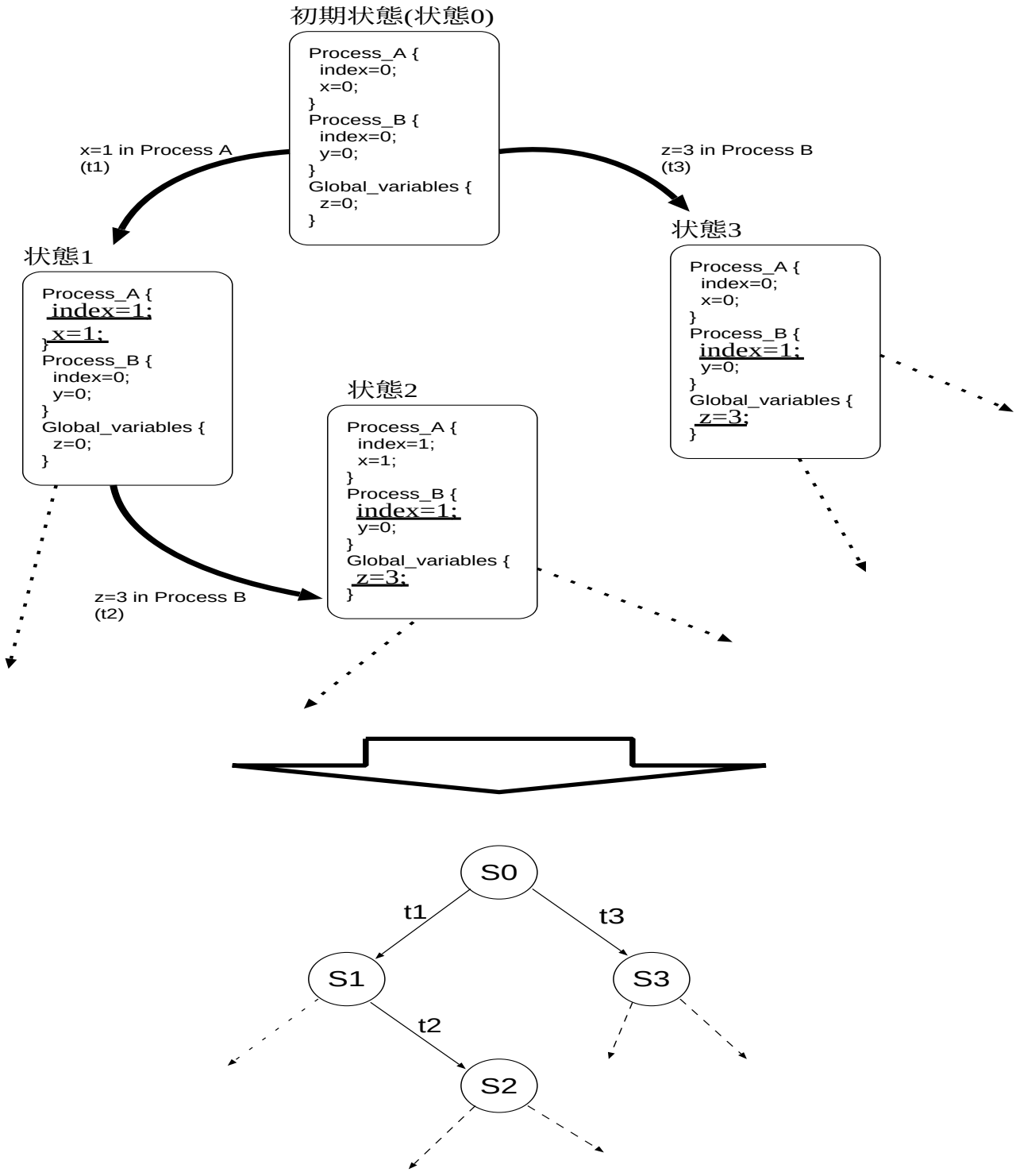
初期状態(状態0)
Process_A {
index=0;
x=0;
}
Process_B {
index=0;
y=0;
}
Global_variables {
z=0;
}
x=1 in Process A
(t1)
z=3 in Process B
(t3)
状態1
Process_A {
index=1;
x=1;
}
Process_B {
index=0;
y=0;
}
Global_variables {
z=0;
}
状態2
Process_A {
index=1;
x=1;
}
Process_B {
index=1;
y=0;
}
Global_variables {
z=3;
}
z=3 in Process B
(t2)
状態3
Process_A {
index=0;
x=0;
}
Process_B {
index=1;
y=0;
}
Global_variables {
z=3;
}
S0
t1
t3
S1
S3
t2
S2

図 2.2: 状態空間探索

例えば，共有変数 v, w を持ち，以下のトランジションを含む逐次プロセス X, Y から構成される並行プロセスが存在したとする．

プロセス X		プロセス Y	
t_{a0}	$v := 1$	t_{b0}	$v := 2$
t_{a1}	$w := 3$	t_{b1}	$w := 4$

この時，

依存関係がある $(t_{a0}, t_{b0}), (t_{a1}, t_{b1}), (t_{a0}, t_{a1}), (t_{b0}, t_{b1})$

依存関係が無い $(t_{a0}, t_{b1}), (t_{a1}, t_{b0})$

となる．

2.6 スリープセット

スリープセットを用いた探索について解説する．準備として幾つかの用語を定義する．

トランジション系列 σ において，σ に含まれる隣り合った依存関係のないトランジションを交換する操作を繰り返す事によって生成されるトランジション系列の集合をトレースと呼ぶ．σ において，初期状態からの実行系列が同じ状態を 2 回通らない場合，σ が非循環であると言う．トレースに含まれる全てのトランジション系列が非循環であるトレースを，非循環なトレースと言う．

スリープセットは空集合を含むトランジションの集合であり，状態と共に保持される．状態 s からトランジション t により新たな状態 s' を生成した際，s が持つスリープセットに含まれるトランジションのうち t と依存関係が無いトランジションの集合を s' のスリープセットとして継承する．s' からバックトラックにより s に戻った際，t が s のスリープセットへ追加される．スリープセットに含まれるトランジションは実行されない．

図 2.3は，2 つの逐次プロセスからなる並行プロセスの，スリープセットを用いた状態空間探索の様子である．2 個のローカルプロセスから構成される並行プロセスで，状態の中に書かれている番号は，状態が探索される順序である．初期状態 1 から状態 5 まで深さ優先で探索を行い，バックトラックして状態 2 まで戻る．バックトラックした際に状態 2 のスリープセットにトランジション $t2$ が追加される．このスリープセットは状態 2 から新たに生成された状態 6 へと引き継がれ，状態 6 ではトランジション t_2 は実行されない．

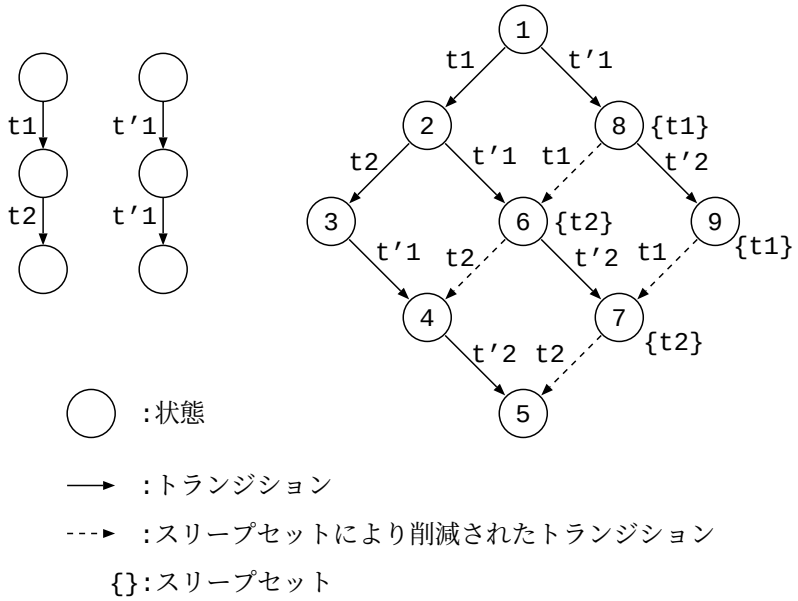

図 2.3: スリープセットを使用した状態空間探索

次の状態 7 も同様である．同様に，状態 8，9 にはスリープセットに t_1 が含まれ実行されない．

トレースが非循環の場合，スリープセットを用いた探索では，トレースに含まれるトランジション系列はただ 1 つの系列が実行される [1]．従って，生成される状態空間に含まれる状態の数を削減することが可能となる．本稿では，スリープセットを併用した状態空間探索について議論する．

第 3 章

キャッシング

本章では，状態空間探索の手法であるキャッシングについて解説する．

並行プロセスの状態空間探索では，並行性により生成される状態の数が膨大となる．これを状態空間爆発と呼ぶ．状態空間探索では，生成された状態空間を全て保存し，新たに探索した状態が既に探索済みか判別しながら探索を行う．しかし，状態空間爆発により生成された状態空間を完全に保存する事は困難である．この為，状態空間の一部のみを高速な主記憶に保存する事により，探索時間の短縮を図るキャッシングと呼ばれる手法が提案されている [1].

状態空間を保存する主記憶はキャッシュと呼ばれる．キャッシングを用いた探索では，キャッシュが一杯になるまでは通常の手法と同様，探索した状態を逐次キャッシュに保存する．キャッシュが一杯になった場合，キャッシュ中から状態を選択して削除し，探索を続行する．

3.1 ダブルワーク

全ての状態空間を保存しないキャッシングを用いた探索では，キャッシュに保存されていない状態に到達した場合にはその状態が探索済みと判断が出来ない．その状態が既に探索された状態の場合，一度実行されたトランジション系列を再度探索する事になる．これをダブルワークと呼ぶ.

探索時間の短縮には，ダブルワークの発生を抑制する事が有効である．キャッシュに再度到達する状態を優先して保存する為には，キャッシュから削除する状態を決定する方法が重要となる.

ダブルワークについては，5章にて詳しく解説する.

most frequently 訪れた回数が最も多い状態を削除する．

lease frequently 訪れた回数が最も少ない状態を削除する．

largect class 訪問した回数でクラス分けを行い，最も大きいクラスの中から状態を削除する．

blind, round-robin selection キャッシュを円状のバッファとして扱う．ポインタが指し示す状態を盲目的に削除し，円の中で，次の状態にポインタを一つ進める．

smallest subtree 深さが最も深い状態 (=最も小さな部分木のルートノード) を削除する．

表 3.1: 削除する状態の決定方法

1. プログラムの制御構文
2. メッセージの受信
3. メッセージの送信
4. 変数割り当て
5. その他

表 3.2: トランジションの種類 (1 がキャッシュに残す優先順位が高い)

3.2 削除状態の決定方法

実験的に検証された，キャッシュからの削除状態の決定方法を表 3.1に示す．幾つかの並行プロセスで実験的に検証した結果，表 3.1の中では，round-robin selection がプロセスの種別に寄らず探索時間が短いとされている [3].

[1] では，到達したトランジションの種別によって状態をクラス分けし，表 3.2の順に優先付けしてキャッシュに保存する方法を提案している．削除する状態を探す際，優先順位が低いクラスに属する状態から順に探して見付かった場合は削除する．見付からない場合には優先順位を 1 つ上げ，同様に繰り返す．round-robin selection より探索時間が短かい事が実験により示されている．

第 4 章

並行プロセスのモデル化

ダブルワークの発生を理論的に検証する為，並行プロセスのモデル化を行う．ラベル付き遷移システム (LTS) により並行プロセスを表現する．

4.1 並行プロセス

n 個の逐次プロセスからなる並行プロセス $P = P_i(i = 1, \ldots n)$ を構成する，各 P_i のラベル付き遷移システムによるモデルは，

$$P_i = (A_i, S_i, \Delta_i, s_{i0})$$

となる．ここで，

- A_i:動作ラベルの集合
- S_i:(ローカル) 状態の集合
- $\Delta_i \subset S_i \times A_i \times S_i$:状態遷移関係
- $s_{i0} \in S_i$:初期状態

である．

このモデルの意味はラベル付き遷移システム $S_P = (A, S, \Delta, s_0)$ により記述される．

- $A = \cup_{i=1,n} A_i$
- $S = S_1 \times \ldots \times S_n (s \in S$ の第 i 成分を $s(i)$ で表す．$)$

- $s_0 = (s_{10}, \ldots, s_{n0})$
- $(s, a, s') \in \Delta \Leftrightarrow$

$$\forall i \in \{1, \ldots, n\} : \begin{cases} (s(i), a, s'(i)) \in \Delta_i & (a \in A_i) \\ s(i) = s'(i) & (a \notin A_i) \end{cases}$$

この意味モデルでは，共通のラベルを持つ動作は「同期して」実行される事を表現している．

4.2 トランジション

動作ラベルと以下に定義するトランジションを区別して扱う．トランジションとは関係するプロセスにおいて遷移の前後のローカル状態が等しい様なグローバルな状態遷移の事である．

まず，Δ 上に同値関係 $\simeq$ を定義する．動作ラベルが等しい 2 つの状態遷移

$$(s_1, a, s_1'), (s_2, a, s_2') \in \Delta$$

について，

$$(s_1, a, s_1') \simeq (s_2, a, s_2') \Leftrightarrow \begin{cases} s_1(i) = s_2(i), s_1'(i) = s_2'(i) & (a \in A_i) \\ s_1(i) = s_1'(i), s_2(i) = s_2'(i) & (a \notin A_i) \end{cases}$$

とする．Δ の同値関係 $\simeq$ により生成される各同値類をトランジションという．

- トランジションの集合を T で表す．
- トランジション t の動作ラベルを $A(t)$ で表す．
- 動作ラベルが A_i に含まれる様なトランジションの集合を T_i で表す．すなわち，

$$T_i = \{t \in T | A(t) \in A_i\}$$

- Δ に対し，

$$\underline{\Delta} = \{(s, t, s') | (s, A(t), s') \in t \in T\}$$

を定義する．

- $(s,t,s') \in \underline{\Delta}$ の時，s を t の始点，s' を t の終点と呼ぶ．
- 空トランジション系列

  $$t_\lambda = \{(s,\lambda,s)|s \in S\}$$

  を定義する．
- トランジション系列 $\sigma \in T^*$ に含まれるトランジションの集合を T_σ で表す．
- $\underline{\Delta}^* \subset S \times T^* \times S$ を以下を満たす最小の関係として定義する．

  1. $t_\lambda \in \underline{\Delta}^*$
  2. $(s,\sigma,s') \in \underline{\Delta}^*, (s',t,s'') \in \underline{\Delta} \Rightarrow (s,\sigma t,s'') \in \underline{\Delta}^*$

  Δ^*,Δ_i^* についても同様に定義する．
- $enabled(s) := \{t \in T|\exists s' \in S : (s,t,s') \in \underline{\Delta}\}$.
- トランジションの実行は決定性である．すなわち，任意の $t \in enabled(s)$ について，

  $$|\{s'|(s,t,s') \in \underline{\Delta}\}| = 1$$

  である．従って，次の様な状態遷移関数 $\underline{\delta} : S \times T^* \to S$ を定義出来る．

  $$\underline{\delta}(s,\sigma) := \begin{cases} s' & \text{if}(s,\sigma,s') \in \underline{\Delta}^* \\ undefined & \text{otherwise} \end{cases}$$

  $\underline{\delta}(s,\sigma)$ が定義されている事を $\underline{\delta}(s,\sigma)!$ と書く．
- トランジション系列 $\sigma \in T^*$ に対し，$\psi(\sigma) : T \to \mathbb{N}$ は各トランジションの出現回数を表す写像とする．すなわち，$\psi(\sigma)(t)$ は系列 $\sigma \in T^*$ 中にトランジション t が出現する回数を表す．このとき，トランジションの定義より，直ちに以下が成り立つ．

  補題 4.2.1 $\underline{\delta}(s,\sigma)! \wedge \underline{\delta}(s,\sigma')! \wedge \psi(\sigma) = \psi(\sigma') \Rightarrow \underline{\delta}(s,\sigma) = \underline{\delta}(s,\sigma')$.
- 状態 s から到達可能な状態の集合

  $$\mathcal{R}(s) := \{s' \in S \mid \exists \sigma \in T^* : \underline{\delta}(s,\sigma) = s'\}.$$

- $A_v = \{read(v), write(v, \mathbf{true}), write(v, \mathbf{false})\}$
- $S_v = \{\mathbf{true}, \mathbf{false}\}$
- $\Delta_v = \left\{ \begin{array}{l} (\mathbf{true}, read(v), \mathbf{true}), \\ (\mathbf{false}, read(v), \mathbf{false}), \\ (\mathbf{true}, write(v, \mathbf{true}), \mathbf{true}), \\ (\mathbf{true}, write(v, \mathbf{false}), \mathbf{false}), \\ (\mathbf{false}, write(v, \mathbf{true}), \mathbf{true}), \\ (\mathbf{false}, write(v, \mathbf{false}), \mathbf{false}) \end{array} \right\}$
- $s_{v0} = \mathbf{false}$

表 4.1: ブール値を取る共有変数のラベル付き遷移システムの例

4.3 共有変数

共有変数は $read$，$write$ 等の，その変数へのアクションを動作ラベルとするラベル付き遷移システムとして表現する．表 4.1は，ブール値を取る共有変数 v のラベル付き遷移システム $P_v = (A_v, S_v, \Delta_v, s_{v0})$ の例である．

4.4 依存関係

依存関係はトランジション集合 T 上の 2 項関係として定義される．共有変数をプロセスとしてモデル化することで,「2 つのトランジションに依存関係が存在しないことは，一方が他方の実行可能性を変化させないこと」と同値になる．

4.4.1 依存関係の分類

2 つのトランジションに依存関係があるのは，以下の 2 つの場合である．

- 実行可能なものを実行不能にする (不能依存関係 D_d):

$$(t, t') \notin D_d \ \wedge \ \underline{\delta}(s, t)! \ \wedge \ \underline{\delta}(s, t')! \Rightarrow \underline{\delta}(s, tt')!$$

- 実行不能なものを実行可能にする (可能依存関係 D_e):

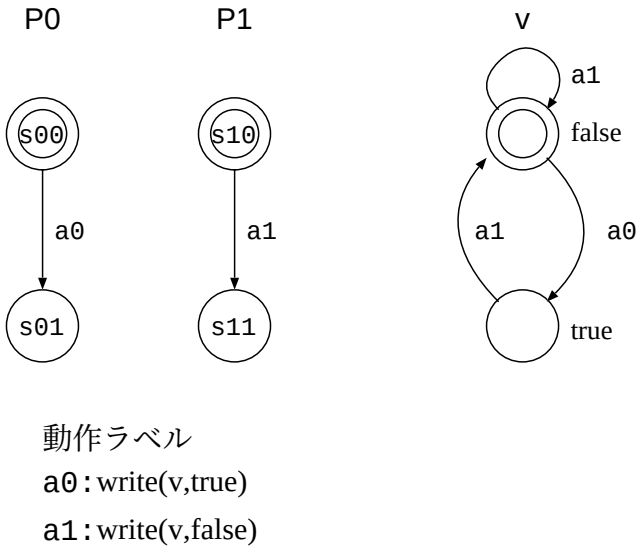

図 4.1: 共有変数をラベル付き遷移システムとして表現した並行プロセスの例

$$(t, t') \notin D_e \ \wedge \ \underline{\delta}(s, tt')! \Rightarrow \underline{\delta}(s, t')!$$

$D_d \cup D_e$ を含む最小の反射的かつ対称的関係を D とする．以下の議論では，依存関係が存在しないトランジション対についての性質を用いる．したがって，依存関係は実際に依存関係のないトランジション対を含んでいても構わない．

4.4.2 依存関係を得るための構文的条件

意味モデル $\mathcal{S}_P = (A, S, \Delta, s_0)$ における不能依存関係 D_d は，並行プログラム P からつぎのようにして得られる．まず，T 上の 2 項関係 D_{br} をつぎのように定義する：$(t, t') \in D_{br}$ であるのは，あるプロセス P_i，状態 $s_1, s_2 \in S$，および，動作ラベル $a, b \in A_i$ が存在して，

$$(s, a, s_1) \in t,\ (s, b, s_2) \in t',$$
$$(s(i), a, s_1(i)), (s(i), b, s_2(i)) \in \Delta_i,$$
$$s_1(i) \neq s_2(i)$$

であるとき，かつこのときに限る．この条件は，あるプロセスにおいて t' と t は実行が競合することを意味する．このとき，

$$D_d = \{(t, t') \in D_{br} \mid \exists s \in S : \underline{\delta}(s, t)! \ \wedge \ \underline{\delta}(s, t')!\}$$

とする．

可能依存関係 D_e はつぎのようにして得られる．まず，T 上の 2 項関係 D_{suc} をつぎのように定義する：$(t,t') \in D_{suc}$ であるのは，あるプロセス P_i，状態 $s_1, s_1', s_2, s_2' \in S$，および，動作ラベル $a, b \in A_i$ が存在して，

$$(s_1, a, s_1') \in t, (s_2, b, s_2') \in t',$$
$$(s_1(i), a, s_1'(i)), (s_2(i), b, s_2'(i)) \in \Delta_i,$$
$$s_1'(i) = s_2(i)$$

であるとき，かつこのときに限る．この条件は，あるプロセス P_i において t' は t の successor であることを意味する．このとき，

$$D_e = \{(t,t') \in D_{suc} \mid \exists s \in S : \underline{\delta}(s, tt')!\}$$

とする．

D_{br} および D_{suc} は，それぞれあるトランジション対が D_d および D_e に属するための十分条件を与える．また，D_{br} および D_{suc} は意味モデル $\mathcal{S}_P$ ではなく，並行プログラム P から直接構成できる．この意味で，構文的（syntactic）条件とよぶ．

4.4.3 D-等価

2 つのトランジション系列は,「隣接する 2 つのトランジション t, t' について，$(t,t') \notin D$ であるときその位置を交換する」という操作を繰り返すことにより一方から他方に変換できるならば，D-等価であるという．トランジション系列 σ と D-等価な系列全体の集合を σ のトレースと言い，$[\sigma]_D$ で表す．

補題 4.4.1 $\underline{\delta}(s,\sigma)! \Rightarrow \forall \nu \in [\sigma]_D : \underline{\delta}(s,\nu)! \wedge \underline{\delta}(s,\sigma) = \underline{\delta}(s,\nu)$.

証明 $\sigma = \sigma_1 tt' \sigma_2$ とし，隣接する 2 つのトランジションの 1 組を考える．$\underline{\delta}(s,\sigma_1) = s'$ とすると，$\underline{\delta}(s', tt')!$ である．

$(t,t') \notin D_e$ より，

$\underline{\delta}(s', t')!$

$(t,t') \notin D_d$ より，

$\underline{\delta}(s', t't)!$ かつ，

$\underline{\delta}(s', t't) = \underline{\delta}(s', tt')$ (補題 4.2.1 より)

従って，

$\underline{\delta}(s, \sigma_1 t t' \sigma_2)!$ かつ，

$\underline{\delta}(s, \sigma_1 t t' \sigma_2) = \underline{\delta}(s, \sigma_1 t' t \sigma_2)$

$[\sigma]_D$ の定義から，この証明を $(t, t') \in D$ であるトランジションの組に対して繰り返し適用する事により，$\forall \nu \in [\sigma]_D$ のトランジション系列に対して帰納的に証明出来る． ∎

4.5 スリープセット

本稿で用いたスリープセットのアルゴリズムは文献 [1] のスリープセットのアルゴリズムの改良で，スタック上に存在する状態の探索は行わないようにしている．

深さ優先による状態空間探索のアルゴリズムを表 4.2に示す．表 4.2にスリープセットを追加した深さ優先探索のアルゴリズムを表 4.3に示す．アルゴリズム中の H はキャッシュ，$Stack$ は深さ優先探索のスタックを表す．また，状態 s のスリープセットは $s.Sleep$ と表し，状態と関連付けて保存される．

アルゴリズム 4.3では，アルゴリズム 4.2にスリープセットの為の処理が追加されている．スリープセットに含まれるトランジションは「T=$enabled(s) - s.Sleep$」により実行されない．状態を生成する場合に「$s'.Sleep = \{t' \in s.Sleep | (t, t') \notin D\}$」により，生成元の状態のスリープセットに含まれるトランジションのうち，状態を生成したトランジションと依存関係の無いものが新たな状態のスリープセットとして引き継がれる．また，バックトラックした際に「$s.Sleep = s.Sleep \cup \{t\}$」で実行したトランジションをスリープセットに追加している．

定理 4.5.1 *アルゴリズム 4.3で* $\mathcal{R}(s_0)$ *のすべての状態が探索される．*

証明 状態 $s \in \mathcal{R}(s_0)$ に対し，初期状態 s_0 から s に到達するトランジション系列の中で，以下のものを w_s で表す（これを状態 s に対する最短最早系列とよぶ）．
(i) s に到達する実行系列で長さが最小．
(ii) (i) の実行系列の中で，アルゴリズム 4.2で最も早く探索される．

w_s はその実行途中で同じ状態を通らない．なぜなら，途中で同じ状態を通れば，s に到達するより短い系列が存在することになるからである．したがって，s に到達するトランジション系列で長さが最小のものはすべてアルゴリズム 4.2で探索される．ゆえに，w_s は s_0 から到達可能な各状態 s に対し必ず存在し，かつ，ユニークである．

w_s がアルゴリズム 4.3でも探索されることを証明する．

$$w_s = t_0 t_1 \cdots t_n$$

とし，また，アルゴリズム 4.2の探索木上で初期状態 s_0 から w_s により状態 s に至るパスを

$$s_0 t_0 s_1 t_1 \cdots t_n s_n = s$$

とする．このパスがアルゴリズム 4.3で探索されないと仮定して矛盾を導く．

探索されないのは，状態 s_i において $t_i \in s_i.Sleep$ となった場合である．このとき，以下のような $s_j,\ j < i$ が存在する．

(i) s_j で t_j を実行するよりも前に t_i が実行され，それがバックトラックしたときに t_i が $s_j.Sleep$ に入る．

(ii) さらに，$s_k,\ j < k \leq i$ で $t \in s_k.Sleep$ である．

このとき，$t_j, \cdots, t_{i-1}$ の各トランジションは t_i と依存関係がないので，s_j から系列 $t_i t_j \cdots t_{i-1}$ が実行可能であり，それにより状態 s_{i+1} に到達する．系列

$$t_0 \cdots t_{j-1} t_i t_j \cdots t_{i-1}$$

はその実行途中で同じ状態を通らない．そうでなければ，w_s より長さが短い s に到達する系列が存在することになり矛盾する．したがって，アルゴリズム 4.2 により系列

$$t_0 \cdots t_{j-1} t_i t_j \cdots t_{i-1} t_{i+1} \cdots t_n$$

は系列 w_s よりも前に探索される．これは矛盾である． ∎

定義 4.5.2 始点 s，終点 s' であるトランジション系列 σ の状態遷移を

$$s \stackrel{\sigma}{\Longrightarrow} s$$

と表す．

定理 4.5.3 アルゴリズム *4.3*において D-等価なトランジション系列集合の中で探索される系列は高々*1* つである．

証明 $\sigma, \nu \in [\sigma]_D$ とし，σ,ν の両方を完全に探索する事は無い事を示す．

$$\begin{aligned} \sigma &= w t w' \\ \nu &= w t' w'' \end{aligned}$$

とし，系列 w 後の状態 s において $t, t' \in enabled(s)$ かつ $(t, t') \notin D$ である．s がスタックに **push** された時点で t, t' のいずれかが $s.Sleep$ に含まれていれば，定理は成立する．ここで，

- t, t' とも $s.Sleep$ に入っていない.
- t が先に実行される.

とし，σ が探索された場合に ν の探索が中断する事を示す．σ,ν の状態遷移を以下の様に置く．

$$\sigma \ : \ s_0 \stackrel{w}{\Longrightarrow} s \stackrel{t}{\Longrightarrow} s_a \Longrightarrow \ldots \Longrightarrow s_b$$
$$\nu \ : \ s_0 \stackrel{w}{\Longrightarrow} s \stackrel{t}{\Longrightarrow} s'' \Longrightarrow \ldots \Longrightarrow s_b$$

1. $[\sigma]_D$ が非循環である場合
   先に探索される σ において，s_a からバックトラックする際に t が $s.Sleep$ に入る．その後 t' が実行され s'' へ向かう．$(t, t') \notin D$ より，$t \in s''.Sleep$ である．

   ν における s'' 以降，t がスリープセットに残り続ければ t は実行される事が無く，定理は成立する．t がスリープセットから除かれる場合は $(t, t'') \in D$ である t'' が t より先に実行される場合のみである．しかし，σ が完全に探索された仮定から，σ では t, t'' の順で実行された事になる．従って，σ と ν で，$(t, t'') \in D$ であるトランジションの実行順序が変わっている．これは，$\sigma, \nu \in [\sigma]_D$ である事と矛盾する．従って，t は s 以降の ν の系列に含まれ，ν の探索は中断する．

2. $[\sigma]_D$ が循環である場合

   - スタックに含まれる状態に向かう循環を含む場合，スタックに達した時点で探索が中断される.
   - スタック以外の状態に向かう循環を含む場合，s_0 から 2 回探索される状態までの，循環を含まないトランジションの系列に対し 1 を適用する．探索が中断されるので，循環を含む部分は探索されない.

■

```
Intialize: Stack is empty; H is enpty;
Search() {
    enter s_0 in H;
    push (s_0) onto Stack;
    DFS();
    }
DFS() {
    s=top(Stack);
    T = enabled(s);
    for all t ∈ T do {
        s' =succ(s)after t;
        if s' ∉ H ∪ Stack {
            enter s' in H;
            push(s') onto Stack;
            DFS();
            }
        }
    pop s from Stack;
    }
```

表 4.2: 深さ優先による状態空間探索

```
Intialize: Stack is empty; H is enpty;
Search() {
    s_0.Sleep = ∅;
    enter s_0 in H;
    push (s_0) onto Stack;
    DFS();
    }
DFS() {
    s=top(Stack);
    T = enabled(s) − s.Sleep;
    for all t ∈ T do {
        s' =succ(s)after t;
        s'.Sleep = {t' ∈ s.Sleep|(t, t') ∉ D};
        if s' ∉ H ∪ Stack {
            enter s' in H;
            push(s') onto Stack;
            DFS();
            }
        s.Sleep = s.Sleep ∪ {t}
        }
    pop s from Stack;
    }
```

表 4.3: スリープセットを追加した状態空間探索

第 5 章

ダブルワークの抑制

本章ではダブルワークが発生する過程を解析し，得られた結果をダブルワークの抑制アルゴリズムに適用する．

5.1 ダブルワークの種類

ダブルワークを以下の 3 種類に分類する．

スリープセットによって抑制可能なダブルワーク D– 等価であるトランジション系列同士は同じ状態へ到達する為，ダブルワークが発生する (図 5.1)．この種類のダブルワークはスリープセットを用いた探索により抑制可能である．

スタック上に保存されている状態に到達するダブルワーク トランジションを実行した結果，スタックに保存されている状態に到達しダブルワークが発生する場合がある (図 5.2)．本稿では，スタックは充分に確保出来ると仮定しているので，このダブルワークは抑制可能である．

スタック外の状態に到達するダブルワーク 図 5.3の様に，トランジション系列を終了するまで探索し，バックトラックして別の系列を実行した結果，既に探索した状態へ到達する場合がある．以降，本稿で扱うダブルワークはこの種類とする．

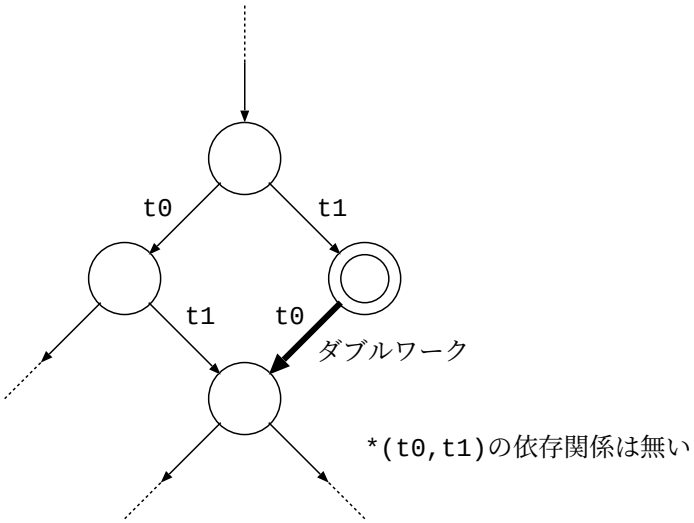

図 5.1: スリープセットにより抑制可能なダブルワーク

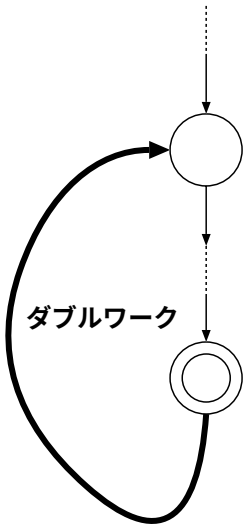

図 5.2: スタック上に到達するダブルワーク

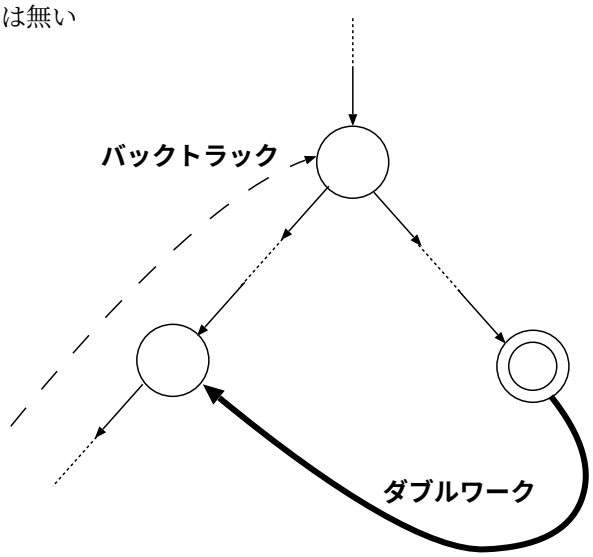

図 5.3: スタック外に到達するダブルワーク

変換規則 1 隣合う依存関係の無いトランジションの順序は交換可能.

変換規則 2 含まれる全てのトランジションが関係するプロセスのローカル状態の遷移が, 始点と終点において等しいトランジション系列 (空列を含む) 同士は交換可能.

表 5.1: トランジション変換規則

5.2 ダブルワーク発生過程の解析

2 つのトランジション系列 σ,σ' を考え，系列 σ が先に探索されるとする．表 5.1のトランジション変換規則を用いて σ を σ' に変換可能な場合，σ,σ' は同じ状態へ到達する．従って，系列 σ' を探索した際にダブルワークが発生する．

定理 5.2.1 変換規則 *5.1*を用いて相互に変換可能な *2*つのトランジション系列は同じ状態へ到達する．

証明

変換規則 1 スリープセットと同じ変換を行う．補題 4.4.1より同じ状態へ到達する．

変換規則 2 状態 s からの 2 つのトランジション系列 σ,σ' を考え，

$$\begin{aligned} \sigma &= w_p w w_s \\ \sigma' &= w_p w' w_s \end{aligned}$$

と置く．w_p を実行した直後までのグローバルな状態遷移は σ,σ' とも等しい．w, w' が変換規則 3 により交換可能であるならば，w, w' が変化させたローカル状態は全て等しくなる．よって，$w_p w$，$w_p w'$ は同じグローバル状態 s' へ到達する．s' 以降のトランジション系列は等しい為，σ,σ' は同じ状態へ到達する．

∎

5.3 ダブルワーク抑制の為のキャッシュ管理方法

状態と共にトランジション系列を保存し，新たな状態を探索する際に系列を比較して互いに交換可能であれば既に探索済みである事が分かる．しかし，最初に状態を探索した時

点で，その状態が再び探索されるか否かを完全に判別する為には，存在するトランジション系列を予め全て調べおかなければならない．これは現実的では無い．

変換規則 5.1の 3 で交換可能な 2 つのトランジション系列のいずれかには goto ジャンプ，ループ，条件分岐構造のいずれかを含む．これらの構造は全て goto ジャンプによって表現可能である．

定理 5.3.1 変換規則 *5.1*の *3*により相互に交換可能な *2*つのトランジション系列 σ,σ' は，いずれかまたは両方に *goto* ジャンプ構造を含む．

証明

$$\begin{aligned}\sigma &= \sigma_p w t \sigma_s \\ \sigma' &= \sigma_p w' t' \sigma_s\end{aligned}$$

と置く．$wt, w't'$ が交換可能であるとし，$T_w \neq T_{w'}$ とする．ローカル状態の遷移を

$$\begin{aligned}s &\overset{w}{\Longrightarrow} s' \overset{t}{\Longrightarrow} s_e \\ s &\overset{w'}{\Longrightarrow} s'' \overset{t'}{\Longrightarrow} s_e\end{aligned}$$

$$t \neq t', s' \neq s''$$

と置く．s'，s'' から到達するローカル状態は同じである為，t，t' の実行後のローカル状態 s_e に含まれる実行位置は同じにならなければならないが，逐次プロセス内の状態遷移の性質より，t，t' のいずれかまたは両方が goto ジャンプ命令でなければならない． ∎

従って，状態の性質として，goto ジャンプを含むトランジション系列によって到達される事が再度訪問される状態の必要条件となる．ここで，

合流ローカル状態 入り次数が 2 以上であるローカル状態.

分岐ローカル状態 出次数が 2 以上であるローカル状態.

と定義する．ローカル状態の初期状態には入り次数を 1 加算する．また，状態の集合 $S_c \subset S$ を次の様に定義する:$s \in S_c$ であるのは，以下を満たす時，かつその時に限る．

1. あるプロセス P_i において $s(i)$ が合流ローカル状態であるか，または，
2. あるプロセス P_i において $s(i)$ は分岐ローカル状態の 1 つ先のローカル状態（すなわち分岐の行き先のローカル状態）である．

S_c に含まれる状態をキャッシュに優先して保存する事でダブルワークを抑制するキャッシュ管理方法を提案する．これは，表 3.2にある Godefroid らの手法 [1] の優先順位の最も高いもののみを扱う事となる．さらに，共有変数をプロセスとして扱う事により Godefroid らの手法に比べてトランジションの種類の判別をより厳密に行う事となる．

トランジション系列 σ の先頭から k 番目の記号を $\sigma[k]$ で表す．

定理 5.3.2 意味モデル $\mathcal{S}_P$ は非循環であるとする．$\sigma, \nu \in T^*$ $(\sigma \neq \nu)$ をつぎのようなトランジション系列とする．

$$\underline{\delta}(s, \sigma) = \underline{\delta}(s, \nu) = s_d.$$

ここで，$s \in \mathcal{R}(s_0)$ であり，また

$$\sigma[1] \neq \nu[1]$$

とする．アルゴリズム *4.3*による探索において，σ が ν よりも先に探索されたとする．このとき，訪問した S_c の状態をすべてキャッシュに保存していれば，系列集合 νT^* の任意の系列の探索は状態 s_d 以前に停止する．

証明 つぎの 2 つの場合が考えられる．

場合 1 ．$(t, t') \in D_{br}$ であるような $t \in T_\sigma$ と $t' \in T_\nu$ が存在する場合．

場合 2 ．それ以外．

まず，場合 2 について考える．

$\mathcal{S}_P$ の非循環性の仮定より，σ と ν は D-等価である．補題 4.5.3より，ν の探索は s_d に到達する前に停止する．すなわち，νT^* の探索は状態 s_d より前に停止する．

つぎに場合 1 について考える．ν を実行中に最後に訪れる S_c の状態を s_v とする．また，$\nu = \nu_1 \nu_2$ に対し，

$$\underline{\delta}(s_0, \nu_1) = s_v,\ \underline{\delta}(s_v, \nu_2) = s_d$$

とする．

σ と ν はあるプロセスで分岐するトランジションを含み，かつ，最終的に到達する状態が同じなので，ν の実行中に必ず合流ローカル状態を訪れる．したがって，このような状態 s_v は必ず存在する．

$s_v = s_d$ ならば σ の探索時に s_d がキャッシュに保存されることになり，νT^* の探索は最悪でも s_v で停止する．

そうでない場合を考える．仮定より，以下が成り立つ．

- ν の実行において s_v のつぎの状態から s_d まで（すなわち ν_2 のトランジションの終点）はローカル状態として合流ローカル状態を含まない．
- ν の実行において s_v から s_d の 1 つ前の状態（すなわち ν_2 のトランジションの始点）はローカル状態として分岐ローカル状態を含まない．

これらのこと，および，σ と ν で到達する状態が s_d で等しいことから，ν_2 のある置換が部分系列として σ に含まれなければならない．

$\sigma[1] \notin T_{\nu_2}$ ならば，

$$\sigma[1]\gamma\nu_2 \in [\sigma]_D$$

であるような系列 γ が存在する．

$$\underline{\delta}(s, \sigma[1]\gamma) = s_v$$

であるから，s_v は ν の探索前にキャッシュに保存され，結果として，νT^* の探索は最悪でも s_v で停止する．

$\sigma[1] \in T_{\nu_2}$ の場合を考える．ν_2 の実行における $\sigma[1]$ の始点のローカル状態は分岐ローカル状態ではないので，任意の $t \in T_\nu - \{\sigma[1]\}$ について $(\sigma[1], t) \notin D_{br}$ である．$\nu = \nu'\sigma[1]\nu''$，$\sigma[1] \notin T_{\nu'}$ とすると，

$$\sigma[1]\nu'\nu'' \in [\nu]_D$$

を証明できる．$\sigma[1]\nu'\nu''$ の探索は ν よりも先に行われるので，補題 4.5.3 により，ν の探索は状態 s_d より前に停止する．すなわち，νT^* の探索は状態 s_d より前に停止する． ∎

S_P が循環を含む場合には，ダブルワークが発生する場合がある．図 5.4は，ローカル状態の循環を含む並行プロセスである．2 個の逐次プロセス P_0，P_1 が存在し，共有変数およびローカル変数は存在しない．依存関係は存在しない．P_0 にはループが含まれていて，トランジション t_2 により，P_0 のローカル状態は s_0 に遷移する．

この並行プロセスでダブルワークは 4 回発生する．そのうち，状態 S_2 と S_5 で t_2 を実行した際に発生するダブルワークは，スタック上の状態を訪問するので防ぐ事が出来る．また，$S_7 = S_4$ はキャッシュに保存される為，S_0 から t_0' により発生するダブルワークは防ぐ事が出来る．しかし，$S_6 = S_5$ はキャッシュに保存されないので，S_1 において t_0' の実行から発生するダブルワークは防ぐ事が出来ない．

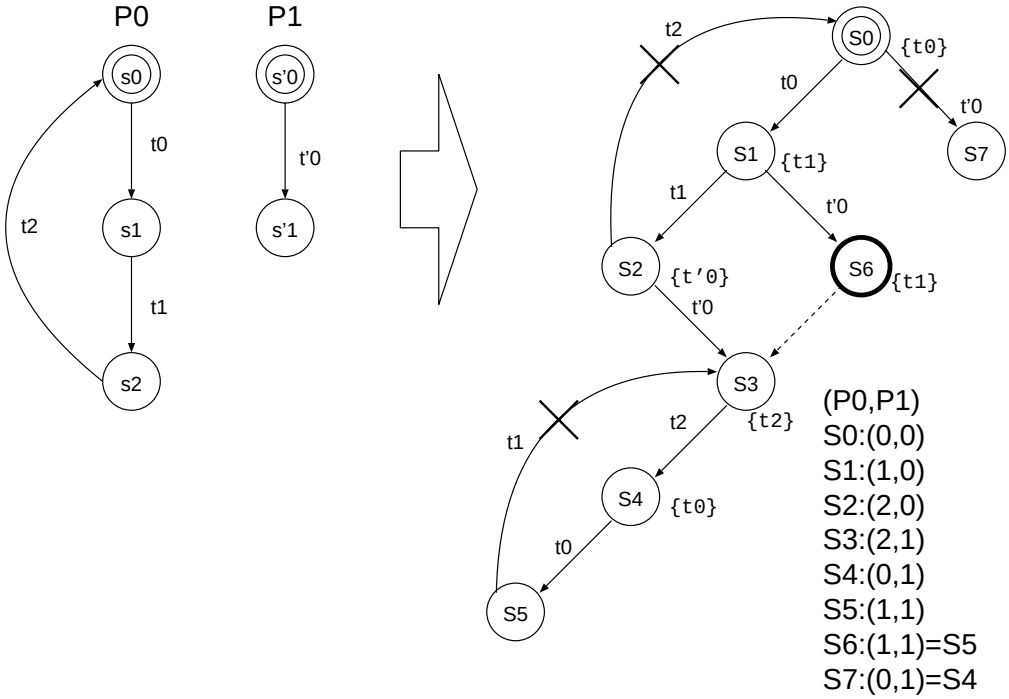

図 5.4: 探索例:循環を含む場合

状態 S_5, S_6 に到達するトランジションをそれぞれ σ,ν と置くと，以下の様になる．

$$\begin{aligned}\sigma &= t_0 t_1 t'_0 t_2 t_0 \\ \nu &= t_0 t'_0\end{aligned}$$

σ を変換規則 5.1で変換する．

$$\begin{aligned}\sigma &= t_0 t_1 t'_0 t_2 t_0 \\ &\Rightarrow t_0 t_1 t_2 t'_0 t_0 \text{ (変換規則 1)} \\ &\Rightarrow t_\lambda t'_0 t_0 \text{ (変換規則 2)} \\ &\Rightarrow t_0 t'_0 \text{ (変換規則 1)}\end{aligned}$$

σ から ν へ変換可能であるので同じ状態へ到達し，ν の実行でダブルワークが発生する．

この場合の $S_6 = S_5$ 以降の探索はスリープセットにより抑制される．スリープセットによって抑制されない場合にはダブルワークを伴う探索が続行するが，通常は循環をもう一度実行するまでに探索は中断する．

図 5.5の並行プロセスには 2 個の逐次プロセス P_0，P_1 と，2 個の共有変数 v，w が存在する．ローカル変数は無い．実行条件があるトランジションは，条件が成立しないとトランジションは実行出来ない．トランジションはそれぞれ以下の通りである．

- P_0

  t_0: $w = 0$ の場合のみ，v をインクリメント

  t_1: $w = 1$ の場合のみ，v に 2 を加算

  t_2: v をインクリメント

- P_1

  t'_2: $w = 0$ か $w = 2$ の場合のみ，w をインクリメント

この並行プロセスの依存関係は，

- 可能依存 $(t_0, t_2)(t_2, t'_0)(t'_0, t_1)$
- 不能依存 $(t_0, t_1)(t_0, t'_0)(t_1, t_0)(t_1, t_2)(t'_0, t_0)(t'_0, t_2)$

この並行プロセスの探索では，トランジション系列 $t_0 t_2 t'_0$ で到達した状態 S_3 が，系列 $t'_0 t_1$ で再び訪問される事により，ダブルワークが発生する．しかし，キャッシュアルゴリズムにより状態 S_2，S_3 はキャッシュに保存される為，ダブルワークを防ぐ事が出来る．

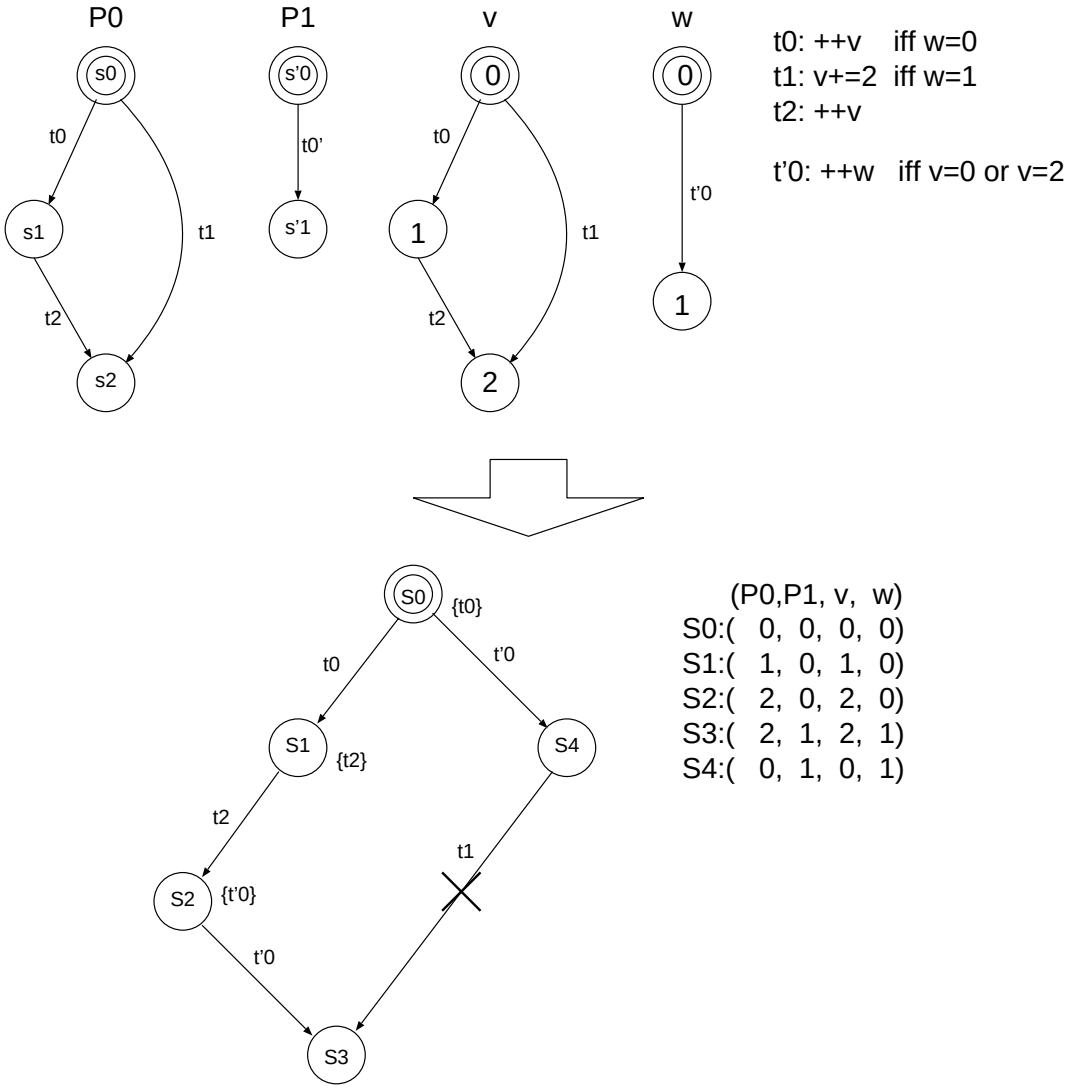

図 5.5: 探索例:循環を含まない場合

第 6 章

結論

並行プロセスから生成される状態空間探索にキャッシングを用いた場合，ダブルワークを抑制する為にキャッシュの管理方法が重要な問題となる．しかしながら，従来法は理論的な考察が加えられておらず，経験的な手法のみが用いられてきた．本研究は，ダブルワーク発生過程の理論的な解析を行い，従来法の解析およびその改良を行った．結果として以下が得られた．

- 定められた規則によるトランジションの変換を行う事により，そのトランジションがダブルワークを発生させるか否かの判別が可能である．
- ダブルワークの発生を抑える為に有効な状態は，入り次数が 2 以上であるローカル状態か，出次数が 2 以上の状態の 1 つ次のローカル状態を含むグローバル状態である．

本研究で得られたダブルワーク発生の性質はトランジションの必要条件である為，そこから導き出された結果から考察したキャッシュ管理方法では，未だ再度探索される事の無い状態を含み，改善の余地がある．今後の課題としては，ダブルワークを発生するトランジションおよび状態の必要十分条件の検証が考えられる．

謝辞

本研究を進めるにあたり熱心に御指導，御教鞭いただいた北陸先端科学技術大学大学情報科学研究科 平石邦彦助教授に心から感謝致します.

参考文献

[1] P. Godefroid, State-Space Caching Revisited, Formal Models in System Design,7,227-241(1995)

[2] G.J.Holzmann, The Model Checker SPIN, IEEE Transactions on Software Engineering, Vol.23, No.5, May 1997

[3] G.J.Holzmann, Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching, IEEE Trans. on Software Engineering, Vol.SE-13, No.6, June 1987

[4] G. J. Holzmann, Tracing Protocols, AT&T Technical Journal,Vol.64,No.10,December 1985