修 士 論 文

シナリオからの並行プログラムの合成についての研究

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

渡邉 裕

1999 年 2 月 15 日

目 次

第 1 章

序論

複数の逐次プログラムが通信を繰り返しながら並行的に動作を行う並行プログラムでは，逐次プログラムのようなデバッグ手法が確立されておらず，そのテスト，デバッグには多大の労力を要する．特に，並行プログラムでは，生成される状態数がプログラムのサイズに対し指数関数的に増大する状態空間爆発の問題があり，全状態を生成して検査するには膨大な時間を必要とする．したがって，並行プログラムの作成段階からできるだけ誤りを含まないようにすべきである．

並行プログラムに特徴的な誤りとして，並行動作の非決定性により生じるものがある．このような誤りは，逐次プログラムのデバッグに比べ，発見することが困難である場合が多い．そこで，あらかじめ期待する正しい動作に対応する並行プログラムの逐次的な実行過程 (これをシナリオと呼ぶ) を与え，それに基づいて並行プログラムを合成することにより，このような誤りの発生を防ぐ手法が内平らによって提案された．この手法が「超逐次プログラミング」[1] である．

「超逐次プログラミング」では，各命令にラベルを付けることにより，プログラムの実行系列を記号列として抽象的に取り扱う．このとき，各命令間には共有変数の $read$, $write$ など，実行順序が結果に影響を与える依存性が存在するが，これは，ラベルの集合上の反射的かつ対称的な関係として与える．まず，並行プログラムに対するシナリオから，それを有向グラフで表現したシナリオグラフを構築する．シナリオグラフの命令間に同期命令を挿入し，各逐次プログラムに射影する．そして，各同期命令について，それを除去しても依存関係に関するシナリオとの等価性を保つならば，すなわちその同期命令が冗長ならば，それを除去する．最後にコード化を行うことにより，期待通りに動作する並行プログラムを得る．

この手法では，冗長な同期命令の削除が試行錯誤的に行われてることから，効率的とは

言えない．また，作成された並行プログラムの最適性についての議論も十分とは言えない．本研究では「超逐次プログラミング」の手法を改良し，冗長な同期命令をほとんど含まず，かつ，正しい動作をする並行プログラムの効率的な合成アルゴリズムを提案する．また，生成される並行プログラムの性質についても考察する．

本論文における構成は次の通りである．第２章では，本研究を進めるにあたり用いられた概念，用語の定義を行う．第３章では提案したアルゴリズムについての説明，第４章でそのアルゴリズムの正当性を検証している．第５章では，座席予約を例とした並行プログラムを，提案したアルゴリズムを用いて合成した実行例を示す．そして，第６章で考察として従来法との比較を行う．最後に結論として合成された並行プログラムの性質，および今後の課題を述べる．

第 2 章

準備

準備として，本研究で用いられるいくつかの概念および用語の定義を行う．

2.1 並行プログラム

並行プログラムとは，図 2.1のように複数の逐次プログラムが互いに通信を繰り返しながら並行的に実行されるようなプログラムのことである．通信は共有変数を介して行われると仮定する．

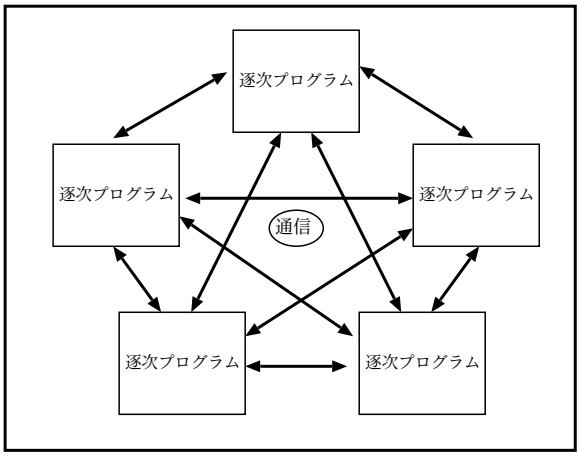

図 2.1: 並行プログラムの構成図

並行プログラムでは各逐次プログラムは並行的に実行されるため，異なる逐次プログラム間の命令の実行順序は確定せず，非決定的である．

例として，図2.2の「命令 a」と「命令 c」に注目すると，プログラマが「命令 c」よりも先に「命令 a」が実行して欲しいと考えていても，個々の逐次プログラムは独立して実行されるため,「命令 c」を先に実行する可能性がある．このように，各逐次プログラムが独立して実行されることから，並行動作の非決定性の問題が生じる．

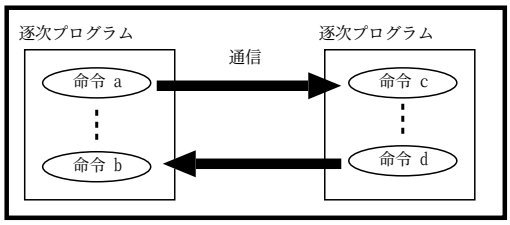

図 2.2: 並行プログラム

2.2 逐次プログラム

並行プログラムを構成する各逐次プログラムについては，以下が成り立つと仮定する．

- 各逐次プログラム単体については，正しい動作をすることが確認されている．すなわち，他の逐次プログラムとの通信が正しく行われると仮定したとき，その逐次プログラムが正しく動作するということが保障されている．
- 各逐次プログラムは，他の少なくとも一つの逐次プログラムと通信を行う．

各逐次プログラムの命令には固有のラベルを付ける．すなわち，ラベルによりそれが対応する命令がどの逐次プログラムのどの命令であるかがわかるようにする．そして，後述するシナリオはこのラベル上の正規表現として与えられる．なお，分岐・ループのような制御構造は，正規表現上では記号 $(+, *)$ により表現されるので，ラベルは分岐・ループの条件文にのみ付ける．

図2.3は2つの逐次プログラムからなる並行プログラムの例である．図中の m1，m2 は共有変数である．

2.3 シナリオ

シナリオとは，プログラマが望む並行プログラムの実行過程であり，つぎのように定義する．

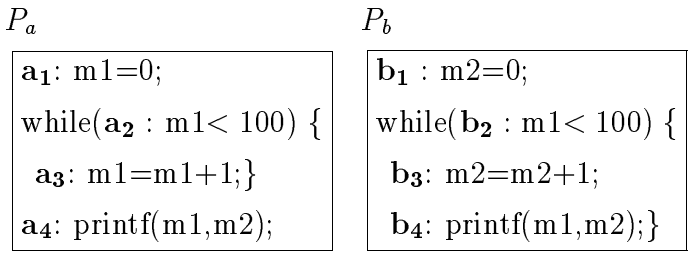

図 2.3: 逐次プログラムの例

- シナリオは，各逐次プログラムの命令に付けられたラベルの集合をアルファベットとする正規表現である．
- シナリオを各逐次プログラムに射影したとき実行可能である．

ここで,「各逐次プログラムについて射影」とは，１つの逐次プログラムに注目し，シナリオからその逐次プログラムに含まれていない記号を全て消去することを意味する．また,「実行可能」とは，通信部分の命令が正しく動作する，すなわち，望むデータが得られたと仮定したとき，その逐次プログラムが正しく動作し，終了することをいう．なお，シナリオを θで表す．ここで，図 2.3の並行プログラムに対するシナリオの例を以下に示す.

例.

$$\theta = a_1 b_1 (a_2 b_2 a_3 b_3 b_4)^* a_4$$

2.4 依存関係

並行動作により実行順序が入れ替わることにより実行結果に何らかの影響を与える可能性のある命令間には依存関係があるという．依存関係は，共有変数を操作する命令や他の逐次プログラムと通信を行う命令間に存在する．たとえば，図 2.3内では，a_1と b_2，a_1と b_4，b_3と a_4等の間に依存関係が存在する．

また，同一逐次プログラム内の命令間についても，それらは定められた順序で実行されることから，これらの命令間の間にも依存関係が存在すると考える．なお，依存関係は各命令につけられたラベルの非順序対 (これを依存関係対と呼ぶ) の集合として定義され，DPで表す.

2.5 同期命令

同期命令とは，異なる逐次プログラムの依存関係のある命令間の実行順序を保つために挿入する命令である．ここでは，セマフォを使った送信命令 (SIGNAL)，受信命令 (WAIT) の組で実現する．この送信命令，受信命令は 1 対 1 に対応している．SIGNAL は，特定の WAIT に向かって信号を送る．WAIT は SIGNAL から信号が送られてくるまで待機し，信号が送られて来た時点で次の命令を実行する．

第 3 章

並行プログラムの合成アルゴリズム

本章では，並行プログラムを合成するアルゴリズムについて説明する．このアルゴリズムは逐次プログラム，シナリオ，および依存関係を入力とし，合成された並行プログラムを出力する．

アルゴリズムは以下のステップから構成されている．

1. シナリオのブロック化
2. 同期命令挿入位置の決定
3. 同期命令の挿入
4. シナリオの復元
5. ループ同期命令の挿入
6. シナリオの各逐次プログラムへの射影
7. コード化

なお，本研究は並行動作によって生じる望ましくない非決定性動作を制限することが目的であり，条件分岐による行き先を制限することは意図しない．実際，条件分岐では変数の値により分岐先が決定されるが，アルゴリズムでは実行系列は各命令に付けられたラベルの系列として表現されるため，変数の値の情報は表現されない．したがって，プログラムの制御構造（分岐およびループ）に対し，以下の仮定を置く．

- 分岐の整合性：可能な分岐先はすべてシナリオ上に表現されている．すなわち，シナリオに含まれるある系列に対し，同じ接頭語を持ち，ある分岐において異なる分岐先に行く実行可能な系列が存在したならば，その系列もシナリオに含まれる．

- ループ脱出の整合性：シナリオ上でのループについて，その脱出条件のチェックは各逐次プロセスごとに行われるが，脱出条件はすべて整合している，すなわち，ある逐次プログラムのループにおいて脱出条件が満たされれば，同じシナリオ上のループを実行している他のすべての逐次プログラムにおいても，同じループの回数で脱出条件が満たされる．

3.1 シナリオのブロック化

一般にシナリオが制御構造を含んでいる場合，その依存関係は複雑になり取り扱いが難しくなる．そこで，これらの構造部分をブロックとして置き換え，制御構造を含む部分と含まない部分に分解する [1]．図 3.1はブロック化した各ブロックについて図解したものである．3.1では，シナリオθ がブロック化されると制御構造 (ループ) を含まないブロックB_0, B_2と含むブロック B_1に分解されている．各々のブロックについて，ブロック B_1を実行することは，B_2を繰り返し実行することであり，B_2が t_jから t_nまで命令の実行である．すなわち，ブロック化しても元のシナリオと同じ意味を持つことなる．また，シナリオへの復元も各ブロックを代入することにより容易にできる．

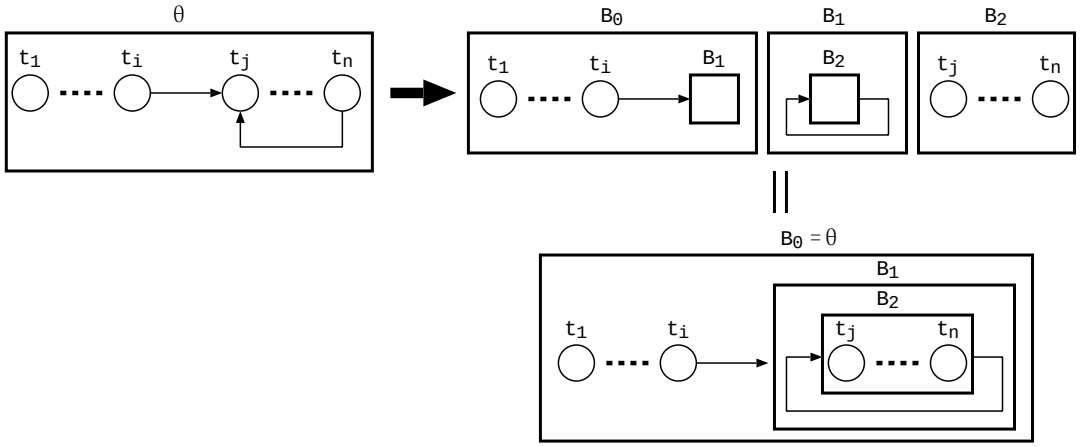

図 3.1: ブロック化

各ブロックは記号，ブロックを表す記号 B_i，$*, +$からなる文字列で構成される．シナリオのブロック化の手順を以下に示す．

1. はじめに，シナリオ全体を１つのブロック B_0とする．

2. 表 3.1の変換表に従ってブロック化を行う.

3. ブロックに置き換えられた部分とそのブロック外の記号間に依存関係がある場合, そのブロックと記号間に依存関係があるものとして，対応する依存関係対（ブロック依存関係対）を依存関係に追加する.

4. 2〜3 を生成したブロック全てに対して行う.

ブロック化した後は，制御構造を含まないブロックに対してのみ同期命令を挿入する.

表 3.1: ブロック化変換表

$B_i = s_m(s_n)^*$	$\Rightarrow$	$B_i = s_m B_j, B_j = (s_n)^*$
$B_i = s_m(s_{n_1} + s_{n_2} + \cdots)$	$\Rightarrow$	$B_i = s_m B_j$ $B_j = s_{n_1} + s_{n_2} + \cdots$
$B_i = (s_n)^*$	$\Rightarrow$	$B_i = B_j^*, B_j = s_n$
$B_i = s_{n_1} + s_{n_2} + \cdots$	$\Rightarrow$	$B_i = B_j + B_k + \cdots$ $B_j = s_{n_1}, B_k = s_{n_2}, \cdots$

以下に，図 2.3に対するのシナリオをブロック化した例を示す.
例.

$$\theta = a_1 b_1 (a_2 b_2 a_3 b_3 b_4)^* a_4$$

$$\Downarrow$$

$$B_0 = a_1 b_1 B_1 a_4,\ B_1 = B_2^*,\ B_2 = a_2 b_2 a_3 b_3 b_4$$

ここで，DPに $(a_1, B_1), (a_4, B_1)$ が追加される.

3.2 同期命令挿入位置の決定

提案アルゴリズムでは，依存関係が存在する命令間の実行順序を固定するために，各逐次プログラム間に同期命令を挿入する．以下に同期命令の挿入位置を決定する手順を示す.

1. 各ブロックからのグラフの作成：各ブロックを構成しているラベルに対応した節点をつくり，次に実行する命令に対応した節点に向かって有向枝を結ぶ.

2. 推移的閉包：グラフの推移的閉包を求める.

3. 枝の除去 (1)：２つの記号間に依存関係がない場合，そのラベルに対応する節点間の枝を除去する．

4. 推移的リダクション：グラフの推移的リダクションを求める．

5. 枝の除去 (2)：同一逐次プログラム内の命令間の枝を削除する．

上記の例で示した B_0 について各手順を行った例を図 3.2に示す．

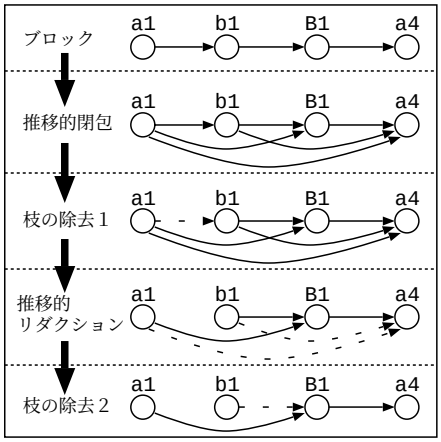

図 3.2: 同期命令挿入位置の決定手順

3.3 同期命令の挿入

有向枝でつながっている節点に対応したラベル間に同期命令を挿入する．同期命令は２つの命令 (SIGNAL，WAIT) から構成されており，SIGNAL を S_n，WAIT を W_n とする．同期命令は，有向枝の始点に対応したラベルの直後に S_n を挿入し，有向枝の終点に対応したラベルの直前に W_n を挿入する．この操作を同期命令挿入位置を決定した全てのブロックに対して行う．

以下に，図 3.2で得られた結果から，B_0 に同期命令を挿入した例を示す．

例．

$$B_0 = a_1 b_1 B_1 a_4 \quad \Rightarrow \quad B_0 = a_1 S_1 b_1 W_1 B_1 S_2 W_2 a_4$$

3.4　シナリオの復元

複数のブロックを１つのシナリオに復元する．復元は以下の例に示すように，各ブロックをそれぞれ代入することにより行う．

例．

$$B_0 = a_1 S_1 b_1 W_1 B_1 S_2 W_2 a_4,\ B_1 = B_2^*,\ B_2 = a_2 b_2 S_3 W_3 a_3 S_4 b_3 W_4 b_4$$
$$\Downarrow$$
$$B_0 = a_1 S_1 b_1 W_1 B_1 S_2 W_2 a_4,\ B_1 = (a_2 b_2 S_3 W_3 a_3 S_4 b_3 W_4 b_4)^*$$
$$\Downarrow$$
$$\theta = a_1 S_1 b_1 W_1 (a_2 b_2 S_3 W_3 a_3 S_4 b_3 W_4 b_4)^* S_2 W_2 a_4$$

3.5　ループ同期命令の挿入

ループ中に異なる逐次プログラムの命令が混在する場合，ループの実行系列の最後に，ループ中に命令が含まれるすべての逐次プログラムが同期するループ同期命令（L_n) を挿入する．

例．

$$\theta = a_1 S_1 b_1 W_1 (a_2 b_2 S_3 W_3 a_3 S_4 b_3 W_4 b_4 \mathbf{L_1})^* S_2 W_2 a_4$$

3.6　シナリオの各逐次プログラムへの射影

復元したシナリオを各逐次プログラムに射影する．このとき，

- 同期命令は，送信命令 (S_n) についてはその記号の直前のラベルが含まれる逐次プログラムに，受信命令 (W_n) についてはその記号の直後のラベルが含まれる逐次プログラムにのみ残す．
- ブロックに対する同期命令は，そのブロックに命令が含まれるすべての逐次プログラムに挿入される．ただし，送信命令と受信命令が同一の逐次プログラムである場合には挿入されない．
- ループ同期命令はループに関わるすべての逐次プログラムに射影される．

例．a_1, a_2は逐次プログラム P_1の命令，b, c はそれぞれ逐次プログラム P_2, P_3の命令とする．

$$\theta = a_1 S_1 W_1 (a_2 b_1 c_1 L_1)^*$$

のとき，θの各逐次プログラム P_iへの射影$\theta|P_i$は

$$\theta|P_1 = a_1S_1(a_2L_1)^*,\ \theta|P_2 = W_1(b_1L_1)^*,\ \theta|P_3 = W_1(c_1L_1)^*$$

となる．$\theta|P_1$では S_1が射影されるため，W_1は射影されない．

以下に，先の例で復元したシナリオを射影した例を示す．

例．

$$\theta = a_1S_1b_1W_1(a_2b_2S_3W_3a_3S_4b_3W_4b_4L_1)^*S_2W_2a_4$$

$$\Downarrow$$

$$\theta|P_a = a_1S_1(a_2W_3a_3S_4L_1)^*W_2a_4$$

$$\theta|P_b = b_1W_1(b_2S_3b_3W_4b_4L_1)^*S_2$$

3.7 コード化

各逐次プログラムの射影からコード化を行う．

例．

P_a

```
a1: m1=0;
S1: SIGNAL(1);
while(a2 : m1< 100) {
 W3: WAIT(3);
 a3: m1=m1+1;
 S4: SIGNAL(4);
 L1: L_SYNCHRO(1)}
W2: WAIT(2);
a4: printf(m1,m2);
```

P_b

```
b1 : m2=0;
W1: WAIT(1);
while(b2 : m1< 100) {
 S3: SIGNAL(3);
 b3: m2=m2+1;
 W4: WAIT(4);
 b4: printf(m1,m2);
 L1: L_SYNCHRO(1)}
S2: SIGNAL(2)
```

図 3.3: コード化した各逐次プログラム

第 4 章

アルゴリズムの正当性

本章では，アルゴリズムにより生成される並行プログラムが以下の性質を持つことを示す．

1. 与えられたシナリオおよび依存関係に対して正しい並行プログラムである．
2. 冗長な同期命令を含まない．

ここで，正しい並行プログラムとは，その任意の実行系列が依存関係の存在しない命令間の順序の入れ換えを除き，シナリオと同じ動作をすることをいう．また，冗長な同期命令とは，削除しても並行プログラムが実現する実行系列全体の集合が変化しないような同期命令のことをいう．

4.1 問題の定式化

提案アルゴリズムでは，逐次プログラムの各命令にラベル付けを行い，命令間の依存関係をラベルの集合上の関係として表現している．これにより，各命令の意味を考慮せずに取り扱うことができる．

いくつかの集合および記法を定義する．

- Σ_i：逐次プログラム P_iの各令に対して付けたラベルの集合．$\Sigma = \bigcup_i \Sigma_i$とする．$\Sigma$上の有限長の系列全体の集合を$\Sigma^*$で表す．
- $s \in \Sigma^*$に対し，s に含まれるΣのラベルの集合を$\Sigma(s)$ で表す．s の長さを $|s|$ で表す．s の i 番目の命令を $s[i]$ で表す．
- $S(\theta)$: Σ上の正規表現であるシナリオ θ が表す実行系列全体の集合．

- $DS(\theta, DP)$: θが表す実行系列と DPの下で依存関係等価 (以下に定義する) になる実行系列全体の集合.
- $PS(P, \theta, DP)$: 逐次プログラムの集合 $P = \{P_1, \cdots, P_n\}$，シナリオ θ，および依存関係 DP に対し，提案したアルゴリズムを適用して得られた並行プログラムの実行系列全体の集合.
- $ct : \Sigma^* \times I_{|s|} \to \Sigma \times I_{|s|}$はつぎのような関数である．ただし，$I_n = \{1, \cdots, n\}$ である.

$$ct(s, i) = (s[i],\ s[i] \text{ の同一記号内での出現順位})$$

例． $s = abcbcdef$のとき

$ct(s, 2) = (b, 1),\ ct(s, 4) = (b, 2)$.

また，

$$\Sigma_{ct}(s) = \{ct(s, i) \mid i = 1, \cdots, |s|\}$$

$$ct(s) = ct(s, 1)ct(s, 2) \cdots ct(s, |s|)$$

とする.

定義 4.1.1 先行制約

以下のように定義される$\Sigma_{ct}(s)$ 上の半順序関係を先行制約とよぶ.

$$ct(s, i) \prec_{DP.s} ct(s, j) \Leftrightarrow$$
$$i = j \vee (i < j \wedge ((s[i], s[j]) \in DP \vee s[i] = s[j])).$$

定義 4.1.2 依存関係等価

２つの系列 $s, s' \in \Sigma^*$，および，依存関係対 DPについて以下が成り立つとき，DPの下で s と s'は依存関係等価であるという.

1. $\forall i.\exists j.ct(s, i) = ct(s', j) \wedge \forall j.\exists i.ct(s', j) = ct(s, i)$.
2. $\prec_{DP.s} = \prec_{DP.s'}$.

依存関係等価はΣ^*上の同値関係である．正しい並行プログラムは，その任意の実行系列が $DS(\theta, DP)$ に含まれるものとして定義できる．すなわち，提案アルゴリズムが正しい並行プログラムを出力することは，以下が成り立つことである.

$$PS(P, \theta, DP) \subseteq DS(\theta, DP).$$

もちろん，$DS(\theta, DP)$ に含まれるすべての系列が実行可能であること，すなわち，上式で等号が成り立つことが望ましい.

4.2 制御構造を含まないシナリオ

制御構造が含まれていないシナリオから並行プログラムを合成する場合について考える．まず，合成された並行プログラムが正しい並行プログラムであることを証明する．さらに，制御構造を含んでいないシナリオに対しては，任意の正しい実行系列はアルゴリズムにより作られた並行プログラムの実行系列になり得ることも示す．すなわち，

$$PS(P, \theta, DP) = DS(\theta, DP)$$

であることを示す．

アルゴリズムではシナリオ θをブロック化した後，各ブロックごとにグラフを作成する．しかし，制御構造が含まれていない場合，シナリオ θ は１つの実行系列を表すので（$S(\theta) = \{\theta\}$），ブロック化を行っても全体が１つのブロックに置き換わるだけであり，作成されるグラフは１つである．これを $G_1(\theta)$ とする．グラフ $G_1(\theta)$ は各節点が $\Sigma_{ct}(\theta)$ の要素に対応しており，有向枝がつぎに実行する命令に対応する節点を指すような有向非循環グラフである．θにおいて同一命令はたかだか１回しか出現しないことから，$\Sigma_{ct}(\theta) = \{(\sigma, 1) \mid \sigma \in \Sigma(\theta)\}$ となる．すなわち，$\Sigma_{ct}(\theta)$ と$\Sigma(\theta)$ の各要素は１対１に対応する．

つぎにグラフ $G_1(\theta)$ の推移的閉包 $G_2(\theta)$ を求める．$G_1(\theta)$ は１つの有向パスから構成されるグラフとなるため，$G_2(\theta)$ の任意の２節点間にはシナリオが表す実行系列の順序に従った有向枝のみが存在する．そして，依存関係の存在しない節点間の有向枝の除去を行いグラフ $G_3(\theta)$ を得る．$G_3(\theta)$ では，依存関係が存在する命令間については実行順序を示す有向枝が残る．この後，推移的リダクションを行い，グラフ $G_4(\theta)$ を得る．最後に，同一逐次プログラムの節点間の有向枝の除去を行いグラフ $G_5(\theta)$ を得る．$G_5(\theta)$ の有向枝に対応する同期命令を各逐次プログラムに挿入する．

グラフ $G_4(\theta)$ からつぎのような $\Sigma_{ct}(\theta)$ 上の半順序関係$\prec_{G_4(\theta)}$が得られる．

$ct(\theta, i) \prec_{G_4(\theta)} ct(\theta, j) \Leftrightarrow$
$(i = j) \vee$ (節点 $ct(\theta, i)$ から節点 $ct(\theta, j)$ への有向パスが存在).

グラフ $G_4(\theta)$ の構成法から，θに出現する任意の２つのラベルσ_i, σ_jについて，$(\sigma_i, \sigma_j) \in DP$ ならば，かつそのときに限り，節点 $(\sigma_i, 1)$ と節点 $(\sigma_j, 1)$ を結ぶ有向パスが存在する．さらに，その有向パスの向きは先行制約$\prec_{DP,\theta}$の順序に従う．このことから，つぎの補題が得られる．

補題 4.2.1 $\prec_{DP,\theta} = \prec_{G_4(\theta)}$.

命題 4.2.2 系列 $s \in \Sigma^*$は，$ct(s)$ がグラフ $G_4(\theta)$ に対するトポロジカルソートの解であるとき，かつ，そのときに限り，$PS(P, \theta, DP)$ に含まれる．

証明． グラフ $G_4(\theta)$ の有向枝は $G_5(\theta)$ で残らなかったものは同一逐次プロセス内での順序関係を表しており，また，$G_5(\theta)$ で残ったものは挿入された同期命令に対応している．

まず，$s \in PS(P, \theta, DP)$ と仮定する．$G_4(\theta)$ の節点 $(\sigma_i, 1)$ から節点 $(\sigma_j, 1)$ への有向枝を考えると，σ_i，σ_jは同一逐次プログラム内の命令ラベルであることにより，または，それらの間に同期命令が挿入されることにより，s においてσ_jの実行はσ_iの実行よりも後になる．このことから，$G_4(\theta)$ の節点 $(\sigma_i, 1)$ から節点 $(\sigma_j, 1)$ を結ぶ有向パスが存在するならば，系列 s におけるσ_jの出現位置はσ_iよりも後になる．すなわち，$ct(s)$ は$\prec_{G_4(\theta)}$ に対するトポロジカルソートの解である．

逆に，$ct(s)$ が$\prec_{G_4(\theta)}$に対するトポロジカルソートの解であると仮定する．s において依存関係のある命令の出現順序は$\prec_{G_4(\theta)}$に従い，それは各逐次プログラム内での実行順序，および，アルゴリズムにより挿入された同期命令に矛盾しない．また，依存関係の存在しない命令の実行順序は任意のものが実行可能である．したがって，s は合成された並行プログラムで実行可能な系列である． ∎

命題 4.2.2より，つぎの系が得られる．

系 4.2.3 $s \in PS(P, \theta, DP)$ ならば，$\prec_{DP.s} = \prec_{DP.\theta}$である．

つぎに，制御構造が含まれていないシナリオ θに対し，実行系列 $s \in \Sigma^*$が集合 $DS(\theta, DP)$ に属するための条件について考える．シナリオ θおよび依存関係 DPに対し，有向グラフ $G_{DP,\theta} = (\Sigma_{ct}(\theta),\ E_{DP,\theta})$ を定義する．ここで，

$$E_{DP,\theta} = \{(ct(\theta, i), ct(\theta, j)) \mid ct(\theta, i) \neq ct(\theta, j), (ct(\theta, i), ct(\theta, j)) \in \prec_{DP.\theta}\}$$

である．$G_{\theta,DP}$は有向非循環グラフになる．また，先行制約は推移的であるから，グラフ $G_{\theta,DP}$の 2 つの異なる節点 $ct(\theta, i), ct(\theta, j)$ に対し，$ct(\theta, i)$ から $ct(\theta, j)$ への有向パスが存在するとき，かつそのときに限り，それらの間に先行制約 $ct(\theta, i) \prec_{DP.\theta} ct(\theta, j)$ が存在する．依存関係等価の定義よりただちにつぎの命題が得られる．

命題 4.2.4 系列 $s \in \Sigma^*$は，$ct(s)$ がグラフ $G_{\theta,DP}$に対するトポロジカルソートの解であるとき，かつ，そのときに限り，$DS(\theta, DP)$ に含まれる．

グラフ $G_4(\theta)$ とグラフ $G_{\theta,DP}$は同じ節点集合をもち，また，補題 4.2.1よりそれらの推移的閉包は一致する．このこと，および，命題 4.2.2，4.2.4よりつぎの定理が得られる．

定理 4.2.5 $PS(P,\theta,DP) = DS(\theta,DP)$.

つぎに，アルゴリズムで生成される並行プログラムには冗長な同期命令が含まれないこと，すなわち，削除しても並行プログラムが実現する実行系列全体の集合が変化しないような同期命令が存在しないことを示す.

アルゴリズムで生成された並行プログラムに含まれる同期命令を任意に選ぶと，それに対応する有向枝がグラフ $G_4(\theta)$ に存在する．$G_4(\theta)$ からその有向枝を取り除いたグラフを $G'_4(\theta)$ とする．$G_4(\theta)$ は $G_3(\theta)$ の推移的リダクションであることから，$G_4(\theta)$ と $G'_4(\theta)$ の推移的閉包は異なり，さらに，$\prec_{G'_4(\theta)} \subset \prec_{G_4(\theta)}$となることから，$G'_4(\theta)$ に対するトポロジカルソートの解集合は $G_4(\theta)$ のそれを真に含む．したがって，任意の同期命令は冗長ではない.

また，つぎの結果により，同期命令の挿入位置も一意に定まる.

定理 4.2.6 任意の有向非循環グラフについて，その推移的リダクションは一意である [9].

以上の考察により，提案アルゴリズムにより挿入される同期命令は最適なものであるといえる.

4.3 制御構造を含むシナリオ

制御構造を含むシナリオ θは，ブロック化により正規表現の記号+または∗を含むブロック（+–ブロック，∗–ブロック）と含まないブロック（制御構造なしブロック）に分けられる．このとき，ブロックを表す記号 B_iを含んだブロック依存関係対を DPに追加したものを$\widehat{DP}$で表す．すなわち，

$$\begin{aligned}&\widehat{DP} = DP \cup \\ &\{(\sigma_i, B_k) \mid \exists \sigma_j \in \Sigma(B_k) : (\sigma_i, \sigma_j) \in DP\} \cup \\ &\{(B_k, \sigma_j) \mid \exists \sigma_i \in \Sigma(B_k) : (\sigma_i, \sigma_j) \in DP\} \cup \\ &\{(B_k, B_r) \mid \exists \sigma_i \in \Sigma(B_k), \sigma_j \in \Sigma(B_r) : (\sigma_i, \sigma_j) \in DP\}.\end{aligned}$$

ここで，$\Sigma(B_k)$ はブロック B_kに含まれるラベルの集合である．各制御構造なしブロックに対し，制御構造なしの場合と同様な方法で同期命令を挿入する.

まず，生成された並行プログラムが正しい，すなわち，$PS(P,\theta,DP) \subseteq DS(\theta,DP)$ であることを証明する．つぎに，等号が必ずしも成り立たないことを例により示す.

$s \in PS(P,\theta,DP)$ を与えれば，各逐次プログラム内での分岐およびループに対し，分岐先およびループを回る回数を一意に決定できる．さらに，仮定（分岐の整合性，および，

ループ脱出の整合性）により，シナリオ θ上での分岐先およびループの回数を一意に決定できる．

$s \in PS(P, \theta, DP)$ に対し，以下の操作を行う．

1. 各+–ブロックは，s における分岐先に従って他の分岐先を消去する．すなわち，$B_i = B_{i1} + \cdots + B_{ij} + \cdots + B_{in}$について，$s$ において B_{ij}が選択されたならば $B_i = B_{ij}$とする．

2. 各∗–ブロックは，s におけるループの回数に従って展開する．すなわち，$B_i = B_j^*$について，s においてこのループを r 回回るならば

$$B_i = B_j^1 B_j^2 \cdots B_j^r,\ B_j^k = B_j (k = 1, \cdots r)$$

とする．

なお，アルゴリズムでは各 B_j^iの実行の後に，ループの関係する全てのプロセスが同期する命令（ループ同期命令）が挿入される．したがって，B_j^{i+1}の命令の実行は B_j^iのすべての命令が実行された後になる．

3. 下位ブロックを上位ブロックに代入することにより，制御構造なしのシナリオ s_θが得られる．

4. 制御構造のない場合のアルゴリズムの適用により，s_θからグラフ $G_1(s_\theta)$ および $G_2(s_\theta)$ を作る．$G_2(s_\theta)$ からブロックを含む依存関係$\widehat{DP}$が存在しない節点間の有向枝を削除したグラフを$\hat{G}_3(s_\theta)$ とする．すなわち，有向枝 $(ct(s_\theta, i), ct(s_\theta, j))$ は以下のすべての条件を満たさないときに限り削除される．

   (a) $(s_\theta[i], s_\theta[j]) \in DP$.

   (b) $ct(s_\theta, i)$ がある∗–ブロック B_jを展開した B_j^iに属し，また，$ct(s_\theta, j)$ が B_j^{i+1}に属する．

   (c) $ct(s_\theta, j)$ を含むブロック B_kが存在し，$(s_\theta[i], B_k) \in \widehat{DP}$.

   (d) $ct(s_\theta, i)$ を含むブロック B_kが存在し，$(B_k, s_\theta[j]) \in \widehat{DP}$.

   (e) $ct(s_\theta, i)$ を含むブロック B_kおよび $ct(s_\theta, j)$ を含むブロック B_rが存在し，$(B_k, B_r) \in \widehat{DP}$.

5. $\hat{G}_3(s_\theta)$ の推移的リダクションを$\hat{G}_4(s_\theta)$ とする．

$\hat{G}_4(s_\theta)$ により定義される半順序関係$\prec_{\hat{G}_4(s_\theta)}$は，以下に説明するようにアルゴリズムが出力する並行プログラムにおいて保たれる．

まず，(b) の先行制約はループ同期命令の挿入により保たれる．また，(c)〜(e) の先行制約は，ブロック依存関係対の存在により対応する同期命令が挿入され保たれる．(a) の先行制約については，$ct(s_\theta, i)$，$ct(s_\theta, j)$ が共にあるブロック内の命令ならば，対応する同期命令の挿入により保たれる．そうでない場合でも，これらの命令を含むブロックに対する依存関係対が$\widehat{DP}$に追加されることにより先行制約が保たれる．例えば，$ct(s_\theta, j)$ がブロック B_kに含まれ，$ct(s_\theta, i)$ と B_kが別のブロック B_rに含まれているならば，依存関係対 $(s_\theta[i], B_k)$ が$\widehat{DP}$に加えられることになり，対応する同期命令の挿入により $ct(s_\theta, i)$ と $ct(s_\theta, j)$ に関する先行制約が保たれる．

これらのことから，以下が成り立つ．

補題 4.3.1 $s \in PS(P, \theta, DP)$ ならば，$\hat{G}_4(s_\theta)$ に対するトポロジカルソートの解はすべて $PS(P, \theta, DP)$ に属する．

$s \in S(\theta)$ に対し，制御構造を含まない場合のアルゴリズムを適用してグラフ $G_4(s)$ を作る．このとき，命題 4.2.4と同様に以下が成り立つ．

補題 4.3.2 $s \in S(\theta)$ のとき，$G_4(s)$ に対するトポロジカルソートの解はすべて $DS(\theta, DP)$ に属する．

定理 4.3.3 $PS(P, \theta, DP) \subseteq DS(\theta, DP)$.

証明． 任意の $s \in PS(P, \theta, DP)$ に対し，シナリオ $s_\theta \in S(\theta)$ が一意に作れる．s は$\hat{G}_4(s_\theta)$ に対するトポロジカルソートの解の 1 つである．グラフ$\hat{G}_4(s_\theta)$ とグラフ $G_4(s_\theta)$ を比較すると，グラフの構成法から，$\hat{G}_4(s_\theta)$ の枝集合は $G_4(s_\theta)$ の枝集合を含む．これは，$G_4(s_\theta)$ に対するトポロジカルソートの解集合が$\hat{G}_4(s_\theta)$ に対するそれを含むことを意味する．したがって，補題 4.3.1, 4.3.2より定理が得られる． ∎

$DS(\theta, DP) - PS(P, \theta, DP) \neq \emptyset$ であるような場合は存在する．つぎの例を考える．

$$\theta = a_1(a_2b_1b_2)^*, DP = \{(a_1, b_2), (a_1, a_2), (b_1, b_2)\}$$
$$\Downarrow$$
$$B_0 = a_1B_1,\ B_1 = B_2^*,\ B_2 = a_2b_1b_2$$

この場合，$\widehat{DP} = DP \cup \{(a_1, B_1)\}$ となる．ここで，b_1に注目すると，(a_1, B_1) に対応する同期命令の挿入により a_1よりも先に実行されることはない．したがって，$DS(\theta, DP)$ には実行系列 $b_1a_1a_2b_2$が存在するが，$PS(P, \theta, DP)$ には存在しない．

つぎに，制御構造が存在する場合，冗長な同期命令は存在しないかについて考える．ここで，以下に制御構造を含むシナリオの例を示して考察する．

例．a_1, a_2は逐次プログラム P_aの命令，b, c はそれぞれ逐次プログラム P_b, P_cの命令とする．また，依存関係 $DP = \{(a_1, b), (a_2, c), (b, c), (a_1, a_2)\}$ とする．

$$\theta = a_1(bc)^*a_2$$

このようにシナリオ，依存関係が与えられたとき，同期命令挿入後の各々の逐次プログラムについて射影した結果，次のようになる．

$$\theta|P_a = a_1S_1W_2a_2$$
$$\theta|P_b = W_1(bS_3L_1)^*S_2$$
$$\theta|P_c = W_1(W_3cL_1)^*S_2$$

ここで，各逐次プログラムの射影に注目すると，$\theta|P_c$内の W_1は a_1が c がより先に実行するように挿入された同期命令であるが，W_1を除去しても，$a_1 \prec_{\theta.DP} b(S_1, W_1), b \prec_{\theta.DP} c(S_3, W_3)$ により，c は a_1よりも常に後に実行されることになり，$PS(P, \theta, DP)$ が変化することがない．すなわち，この W_1は冗長な同期命令になる．ただし，$\theta|P_b$の W_1を除去すると，先行制約が保たれなくなるため，冗長ではない．

また，a_1と c の間には先行制約が無いため，同期命令を挿入する必要がないということでも W_1は冗長な同期命令であるといえる．

このように，ブロック化した部分に対し，複数の逐次プログラムに１つの同期命令を射影することにより，いずれかの逐次プログラムには冗長な同期命令が挿入されることがあるが，挿入した同期命令そのものが冗長にはならない．したがって，必ずしも合成された並行プログラムには冗長な同期命令は含まれないとは言えない．

第 5 章

実行例

本章では，座席予約の問題を用いて提案したアルゴリズムを実行した例を示す.

5.1 問題設定

２つの逐次プログラム P_a, P_bがそれぞれ独立に座席を予約し，予約結果を表示する並行プログラムを作成する (図 5.1参照)．ただし，２つの逐次プログラムは同時に実行され，座席予約は P_aを優先的に行うことにする．

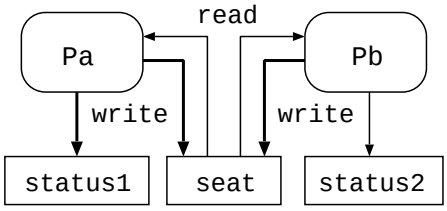

図 5.1: 座席予約の問題

5.2 入力

アルゴリズムの入力となる各逐次プログラムを図 5.2に示す．ただし，seat は共通変数である．そして，READ(i) は外部からの入力を読み取り，i に代入し，関数自体は１を返す．一定時間入力が無い場合は NULL を返す命令である．また，limit() は予約期間内は１，予約期間を過ぎたら０を返す関数である．

各々の逐次プログラムは，予約したい座席の数が全て予約できる (予約したい座席数以上の空席がある) ときのみ予約ができ，空席が無くなったら終了する．しかし，上記の逐

P_a

```
a0: int m;
a1: char *status1;
a2: seat = 100;
while(a3: seat > 0 || limit() != 0) {
 a4: *status1 = "NG";
 if(a5: READ(m) != NULL) {
  if(a6: seat >= m) {
   a7: seat = seat − m;
   a8: *status1 = "OK"; }
  a9: printf("%d %s",m,*status1);}}
```

P_b

```
b0: int n;
b1: char *status2;
while(b2: seat > 0 || limit() != 0) {
 b3: *status2 = "NG";
 if(b4: READ(n) != NULL) {
  if(b5: seat >= n) {
   b6: seat = seat − n;
   b7: *status1 = "OK"; }
  b8: printf("%d %s",n,*status1); }
 b9: printf('seat=%d',seat); }
```

図 5.2: 各逐次プログラム

次プログラムを並行に動作させたとき，空席の数以上に予約されてしまう可能性がある．

次に，シナリオθを以下のように与える．

$$\theta = a_0a_1a_2b_0b_1(a_3b_2b_3a_4(a_5(a_6a_7a_8+\varepsilon)a_9+\varepsilon)(b_4(b_5b_6b_7+\varepsilon)b_8b_9+\varepsilon))^*$$

このシナリオは，それぞれの座席予約の入力に対し，逐次プログラム P_a の予約受付を優先するように作られている．なお，シナリオ中のεは空系列を示している．

依存関係をつぎのように与えられる．

$$DP = \{(a_2, b_5), (a_6, b_6), (a_7, b_5), (a_7, b_9)\}$$

5.3 アルゴリズムの実行

5.3.1 シナリオのブロック化

まず，シナリオθをブロック化する．

$$\theta = a_0a_1a_2b_0b_1(a_3b_2b_3a_4(a_5(a_6a_7a_8+\varepsilon)a_9+\varepsilon)(b_4(b_5b_6b_7+\varepsilon)b_8+\varepsilon)b_9)^*$$

$$\Downarrow$$

$$B_0 = a_0a_1a_2b_0b_1B_1,\ B_1 = B_2^*,\ B_2 = a_3b_2b_3a_4B_3B_4b_9,$$
$$B_3 = B_5+\varepsilon,\ B_5 = a_5B_6a_9,\ B_6 = B_7+\varepsilon,\ B_7 = a_6a_7a_8,$$
$$B_4 = B_8+\varepsilon,\ B_8 = b_4B_9b_8,\ B_9 = B_{10}+\varepsilon.\ B_{10} = b_5b_6b_7$$

制御構造を含まないブロック $(B_0, B_2, B_5, B_7, B_8, B_{10})$ に対して，同期命令の挿入を行う．なお，ブロック化により

$$(a_2, b_5) \in DP \ and \ b_5 \in L(B_1)$$
$$(a_6, b_6) \in DP \ and \ a_6 \in L(B_3) \ and \ b_6 \in L(B_4)$$
$$(a_7, b_9) \in Dp \ and \ a_7 \in L(B_3)$$

となることから $(a_2, B_1), (B_3, B_4), (b_9, B_3)$ を DPに追加する．

5.3.2 同期命令の挿入

各ブロックについて同期命令挿入位置を決定し，同期命令の挿入を行う．ただし，B_7, B_{10} はいずれも同一逐次プログラム内のラベルのみで構成されているため，同期命令の挿入は不要である．また，B_5はブロック B_6と逐次プログラム P_a内のラベルで構成されているが，B_6内のラベルもまた P_aの命令しか含まない．したがって，B_5についても同期命令の挿入は不要である．これは B_8についても同様である．したがって，ここでは B_0, B_2についてのみ同期命令の挿入を行えばよい．

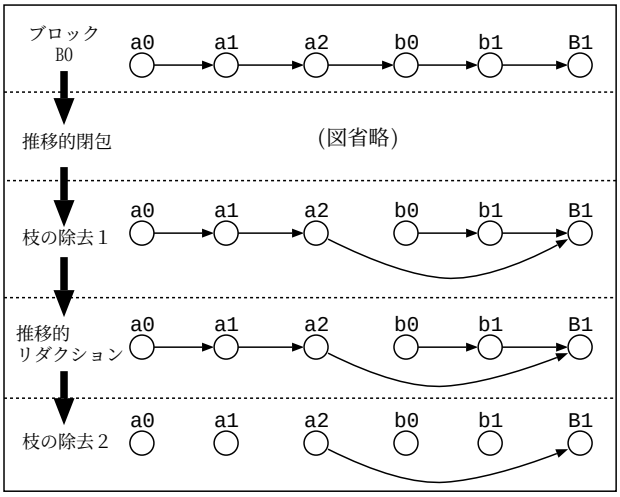

図 5.3: 同期命令の挿入位置決定 (B_0)

図 5.3の結果により，B_0に同期命令を挿入する．

$$B_0 = a_0 a_1 a_2 b_0 b_1 B_1$$
$$\Downarrow$$
$$B_0 = a_0 a_1 a_2 S_1 b_0 b_1 W_1 B_1$$

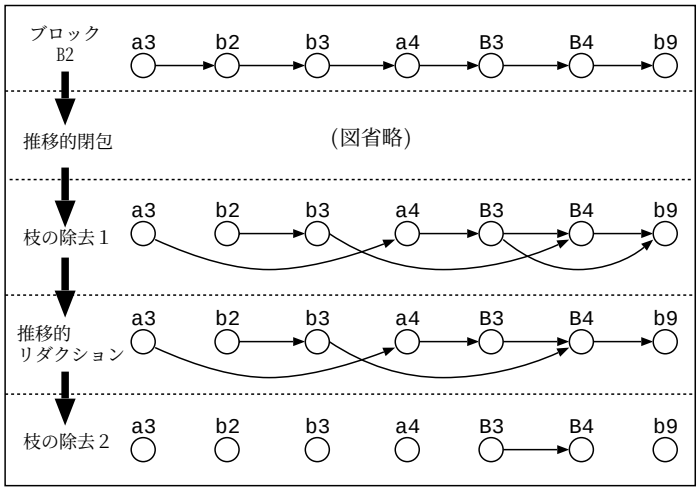

図 5.4: 同期命令の挿入位置決定 (B_2)

同様に図 5.4の結果により，B_2に同期命令を挿入する．

$$B_2 = a_3 b_2 b_3 a_4 B_3 B_4 b_9$$
$$\Downarrow$$
$$B_2 = a_3 b_2 b_3 a_4 B_3 S_2 W_2 B_4 b_9$$

5.3.3 シナリオの復元，射影

同期命令を挿入したブロックを含む全てのブロックからシナリオを復元する．

$$B_0 = a_0 a_1 a_2 S_1 b_0 b_1 W_1 B_1,\ B_1 = B_2^*,\ B_2 = a_3 b_2 b_3 a_4 B_3 S_2 W_2 B_4 b_9,$$
$$B_3 = B_5 + \varepsilon,\ B_5 = a_5 B_6 a_9,\ B_6 = B_7 + \varepsilon,\ B_7 = a_6 a_7 a_8,$$
$$B_4 = B_8 + \varepsilon,\ B_8 = b_4 B_9 b_8,\ B_9 = B_{10} + \varepsilon.\ B_{10} = b_5 b_6 b_7$$
$$\Downarrow$$
$$\theta = a_0 a_1 a_2 S_1 b_0 b_1 W_1 (a_3 b_2 b_3 a_4 (a_5 (a_6 a_7 a_8 + \varepsilon) a_9 + \varepsilon) S_2 W_2 (b_4 (b_5 b_6 b_7 + \varepsilon) b_8 + \varepsilon) b_9)^*$$

シナリオにループが含まれ，かつ，そのループには異なる逐次プログラムの命令が含まれているので，シナリオのループ部分にループ同期命令を挿入する．

$$\theta = a_0 a_1 a_2 S_1 b_0 b_1 W_1 (a_3 b_2 b_3 a_4 (a_5 (a_6 a_7 a_8 + \varepsilon) a_9 + \varepsilon) S_2 W_2 (b_4 (b_5 b_6 b_7 + \varepsilon) b_8 + \varepsilon) b_9)^*$$
$$\Downarrow$$
$$\theta = a_0 a_1 a_2 S_1 b_0 b_1 W_1 (a_3 b_2 b_3 a_4 (a_5 (a_6 a_7 a_8 + \varepsilon) a_9 + \varepsilon) S_2 W_2 (b_4 (b_5 b_6 b_7 + \varepsilon) b_8 + \varepsilon) b_9 L_1)^*$$

復元したシナリオから，各逐次プログラムについて射影を行う.

$$\theta|P_a = a_0a_1a_2S_1(a_3a_4(a_5(a_6a_7a_8 + \varepsilon)a_9 + \varepsilon)S_2L_1)^*$$
$$\theta|P_b = b_0b_1W_1(W_2(b_4(b_5b_6b_7 + \varepsilon)b_8 + \varepsilon)b_9L_1)^*$$

5.3.4 コード化

各逐次プログラムについて射影した結果をコード化した並行プログラムを図 5.5に示す.

P_a

```
a0: int m;
a1: char *status1;
a2: seat = 100;
S1: SIGNAL(1);
while(a3: seat > 0 || limit() != 0) {
 a4: *status1 = “NG”;
 if(a5: READ(m) != NULL) {
  if(a6: seat >= m) {
   a7: seat = seat − m;
   a8: *status1 = “OK” };
  a9: printf(“%d %s”,m,*status1);}
 S2: SIGNAL(2);
 L1: L_SYNCHRO(1);}
```

P_b

```
b0: int n;
b1: char *status2;
W1: WAIT(1);
while(b2: seat > 0 || limit() != 0) {
 b3: *status2 = “NG”;
 W2: WAIT(2);
 if(b4: READ(n) != NULL) {
  if(b5: seat >= n) {
   b6: seat = seat − n;
   b7: *status1 = “OK”; }
  b8: printf(“%d %s”,n,*status1); }
 b9: printf(‘seat=%d‘,seat);
 L1: L_SYNCHRO(1);}
```

図 5.5: 並行プログラム

第 6 章

考察

提案アルゴリズムと内平らによる従来法との違いについて考察する．従来法では，まず，シナリオの集合をシステムのグローバル状態を節点とする有向グラフで表現する．これをシナリオグラフという．そして，以下の手順で同期命令挿入位置を決定する．

1. 依存関係にある直列動作のブロックを並行化するために，シナリオグラフにダミー同期動作を挿入する．
   (a) 依存関係のある直列関係の命令／ブロックの間に同期命令を挿入する．
   (b) 各分岐に同期命令を挿入する．
2. 同期命令を挿入したシナリオを各プロセスへ射影する．
3. 各ブロックごとに冗長な同期命令を抽出しそれを除去する．
   (a) SG を対象とする同期命令を含むシナリオグラフとする．
   (b) 任意の同期命令 s を選択し，それを除去する前のグラフ SG と除去した後のグラフ SG'が等価であれば，同期命令 s を削除する．
   (c) すべての同期命令が除去できなければ，処理を終了する．

以下の例を用いて提案法と従来法の違いを説明する．

- 並行プログラム：

$$P_1 : a$$
$$P_2 : b$$
$$P_3 : c$$
$$P_4 : d$$

- 依存関係：

$$DP = \{(a, b),\ (c, d)\}.$$

- シナリオ：

$$acdb.$$

従来法ではまず，依存関係のある命令間に同期命令を挿入する．

$$acS_1db$$

各プロセスへ射影すると

$$P_1 : aS_1$$
$$P_2 : S_1b$$
$$P_3 : cS_1$$
$$P_4 : S_1d$$

が得られる．ここで，すべての S_1が同時に実行されるようにインプリメントされるとする．$abcd$ は $acdb$ と依存関係等価であるが，この並行プログラムでは生成できない．

提案法ではシナリオから作成したグラフ上で a と b，c と d の間に有向枝が残る．同期命令を挿入し，さらに各プロセスへ射影すると

$$P_1 : aS_1$$
$$P_2 : W_1b$$
$$P_3 : cS_2$$
$$P_4 : W_2d$$

が得られる．この並行プログラムは $abcd$ を生成できる．

この例に示されるように，従来法により合成される並行プログラムでは，たとえ制御構造を含まない場合でも，シナリオに示される正しい動作のすべてが実行可能となるわけではない．これに対し提案手法では，制御構造が含まれない場合については，シナリオに示される正しい動作のすべてが実行可能であることが保証される．

第 7 章

結論

本研究では,「超逐次プログラミング」の手法において，従来，試行錯誤的に行われていた同期命令の挿入について検討し，効率的な挿入位置の決定方法を提案した．この方法を用いて合成される並行プログラムは，以下の性質をもつ．

- 制御構造を含まない場合，シナリオに基づいた正しい動作を行い，かつ，冗長な同期命令を含まない．また，シナリオによって与えられた半順序関係を満たすすべての実行系列を生成できるという意味での最適性も保証される．
- 制御構造を含む場合についても正しい動作を行う．ただし，最適性は必ずしも保証されない．

本研究に対する今後の課題として，以下のことが挙げられる．

- 提案したアルゴリズムにおけるシナリオの役割は非常に重要であり，シナリオにより出力結果が大きく左右される．したがって，効率的に並行プログラムを合成するためのシナリオの与え方についての検討が必要である．
- ブロック化に伴う同期命令の挿入により，プログラムの並行度が低下したり，また，冗長な同期命令が含まれることがある．ブロックに対する同期命令挿入方法の改善，または，ブロック化に替わる新たな手法の提案を考慮する必要がある．

謝辞

本研究を遂行するにあたり，平石 邦彦 助教授に多大なる御指導を頂き，有難うございました．また，同研究室の宋 少秋 助手，高島 康裕 助手には並々ならぬ助言を頂き，深謝しています．

そして，同研究室の皆様には色々な面で御協力頂き有難うございました．本論文で示した諸成果は，お世話になった方々の御指導，ご支援の賜物でございます．この場をおかりし，改めて感謝の意を深く表します．

参考文献

[1] N.Uchihira and H.Kawata, *Scenario-Based Hypersequential Programming: Concept and Example*, IEEE Proceedings,2nd International Workshop on Software Engineering for Parallel and Distributed Systems,pp.277-283,1997.

[2] W.Richard Stevens 著，篠田 陽一 訳, *UNIX* ネットワークプログラミング, トッパン，1992.

[3] David A.Curry 著，アスキー書籍編集部 監訳, *UNIX C*プログラミング, アスキー，1991.

[4] R.E. フィルマン，D.P. フリードマン 共著，雨宮 真人 他共訳, 協調型計算システム：分散型ソフトウェアの技法と道具立て, マグロウヒルブック，1986.

[5] M. ベン-アリ 著，渡辺 栄一 訳, 並行プログラミングの原理：プロセス間通信と同期への概念的アプローチ, 啓学出版，1986.

[6] 五十嵐 善英 著, アルゴリズムと計算可能性, 昭晃堂，1987.

[7] 有川 節夫，宮野 悟 著, オートマトンと計算可能性, 培風館，1986.

[8] V.J.Rayward-Smith 著，吉田 敬一，石丸 清登 訳，井上謙蔵 監修, コンピュータ・サイエンスのための言語理論入門, 共立出版，1986.

[9] A.V. エイホ，J.E. ホップクロフト，J.D. ウルマン 共著，野崎 昭弘，野下 浩平 訳者代表, アルゴリズムの設計と解析 I, サイエンス社，1977.