プログラミング 7－1
（1996. 5. 31）

超逐次プログラミング

— 高信頼並行プログラムの新しい開発手法の提案 —

内平直志　川田秀司　関俊文　本位田真一

uchi@ssel.toshiba.co.jp

東芝 研究開発センター システム・ソフトウェア生産技術研究所
〒210 川崎市幸区柳町70

概要

並行プログラムのいやらしさは，プログラマの想定しなかったタイミングでバグが発生する点にある．プログラマが想定しなかったタイミング（「悪い非決定性」と呼ぶ）は，テストする人も想定できないことが多く，これらのバグの検出は困難であった．我々は，この問題を根本的に解決する並行プログラムの新しい開発手法である「超逐次プログラミング」を提案する．超逐次プログラミングでは，作成した並行プログラムをいったん逐次化し，逐次化されたプログラムに対してテスト・デバッグを行なう．バグが除去できた段階で，並行性を明示的に1つ1つ復元し，最終的な並行プログラムを得る．明示的に並行性を復元しなかった挙動に関しては，最初の逐次化で導入された実行順序がデフォルトで保存される（「デフォルト逐次原理」と呼ぶ）．デフォルト逐次原理により，想定していないタイミング（悪い非決定性）は抑止され，並行プログラムの高信頼化が達成できる．いったん逐次化されたプログラムに対する並行性の復元には，スーパーコンピュータなどで培われた逐次プログラムの並列化技術が利用できる．

Hypersequential Programming

— A Novel Paradigm for Concurrent Programming —

Naoshi Uchihira　Hideji Kawata　Toshibumi Seki　Shinichi Honiden

Systems & Software Engineering Laboratory,
Research & Development Center, Toshiba Corporation
70, Yanagi-cho, Saiwai-ku, Kawasaki-shi, Kanagawa, 210 Japan

This paper proposes *hypersequential programming* which is a novel paradigm for concurrent programming to ease the difficulty of concurrent programming and make the concurrent program highly reliable. The difficulty of concurrent programming is due mainly to its nondeterminism; nondeterminism being the purpose of the concurrent program. We classify nondeterminism into 3 types: *intended*, *harmful*, and *persistent* nondeterminism. In traditional concurrent programming, a programmer first designs and implements programs so as to maximize concurrency, which may include the 3 types of nondeterminism. He then tries to detect harmful nondeterministic behaviors by testing and debugs them. However, it is actually very hard to remove all harmful nondeterministic behavior. On the contrary, in hypersequential programming the concurrent program is first serialized to remove all types of nondeterminism, and then the programmer tests and debugs it as a sequential program. Finally, it is parallelized by restoring only intended and persistent nondeterminism. With hypersequential programming, a highly-reliable concurrent program can be developed because the injection of harmful nondeterminism is precluded. This paper shows the generic concept and embodiments of hypersequential programming.

1 はじめに

一般に，並行プログラムの開発は逐次プログラムの開発より難しい．特に，並行プログラムのデバッグには予定外に苦労させられることが多い [1]．これは，おもに並行プログラムの非決定性によるものである．我々は，非決定性を 3 つに分類した [2].

- 良い非決定性: 設計者が意図した非決定性.
- 悪い非決定性: 設計者が意図しなかった非決定性. すなわちバグ.
- 無害な非決定性: 計算結果には無関係な非決定性.

従来の並行プログラミングでは，プログラマはテストとデバッグにより悪い非決定性を除去していた．しかし，悪い非決定性を 100 %除去するのは現実には困難であった．

本論文では，「超逐次プログラミング (Hypersequential Programming)」と呼ぶ新しいパラダイムを提案する (図 1)．超逐次プログラミングでは，最初にプログラムを逐次化することによりすべての非決定性を除去し，逐次化されたプログラムに対してテスト・デバッグを行なう．次に良い非決定性を明示的に導入し，最後の並行化により無害な非決定性を復元する．従来手法と異なり，超逐次プログラミングでは悪い非決定性が入り込む余地がなく，高信頼な並行プログラムを開発できる．

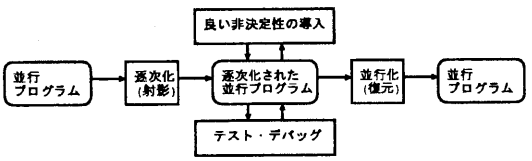


図 1: 超逐次プログラミングの基本コンセプト

2 並行プログラミングの新パラダイム

例を用いて，超逐次プログラミングの基本的コンセプトを説明する．

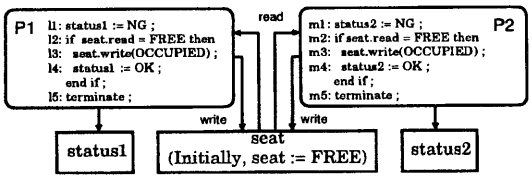


図 2: 並行プログラムの例

2.1 非決定性の分類

まず，図 2の Ada プログラムにおける 3 種類の非決定性を説明する．このプログラムは，「座席予約プログラム」であり，2 つのプロセス P_1, P_2 が共有変数seat をアクセスしている．この並行プログラムは以下の非決定的な挙動 $(\theta_1, ..., \theta_5, ...)$ を持つ.

- $\theta_1 = l_1 \to l_2 \to l_3 \to l_4 \to l_5 \to m_1 \to m_2 \to m_5$
  結果: $status_1 = \mathbf{OK}, status_2 = \mathbf{NG}$.
- $\theta_2 = m_1 \to m_2 \to m_3 \to m_4 \to m_5 \to l_1 \to l_2 \to l_5$
  結果: $status_1 = \mathbf{NG}, status_2 = \mathbf{OK}$.
- $\theta_3 = l_1 \to m_1 \to l_2 \to m_2 \to l_3 \to m_3 \to l_4 \to m_4 \to l_5 \to m_5$
  結果: $status_1 = \mathbf{OK}, status_2 = \mathbf{OK}$.
- $\theta_4 = \mathbf{l_1} \to \mathbf{m_1} \to l_2 \to l_3 \to l_4 \to l_5 \to m_2 \to m_5$
  結果: $status_1 = \mathbf{OK}, status_2 = \mathbf{NG}$.
- $\theta_5 = \mathbf{m_1} \to \mathbf{l_1} \to l_2 \to l_3 \to l_4 \to l_5 \to m_2 \to m_5$
  結果: $status_1 = \mathbf{OK}, status_2 = \mathbf{NG}$.
- .....

良い非決定性: 挙動 θ_1 と θ_2 では結果が異なる (θ_1 では P_1 が予約，θ_2 では P_2 が予約) が，両方とも正しい挙動である．

悪い非決定性: 挙動 θ_3 では，不具合 (ダブルブッキング) が発生する．

無害な非決定性: 挙動 θ_4 と θ_5 では，結果は同じである．これは，l_1 と m_1 はお互いに依存関係がなく，その実行順序が結果に影響しないからである．

2.2 パラダイムシフト

2.2.1 従来パラダイム

従来手法では，並行性をなるべく高めようとして，すべてのタイプの非決定性 $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, ...)$ を含むプログラムをコーディングしてしまう．それから，テストにより悪い非決定性 (θ_3) を検出し，同期命令を用いてプログラムを部分的に逐次化することで悪い非決定性を除去する．しかしながら，すべての悪い非決定性を検出するのは非常に難しく，多かれ少なかれバグは残存する．

この状況を漫画にしたものが図 3である．うっそうとした木は多くのバグを含む並行プログラムを表している．従来手法では，木の枝葉に寄生する虫 (バグ) を 1 つ 1 つ発見しては除去していた．しかし，うっそうとした木の虫をすべて除去するのは不可能であった．

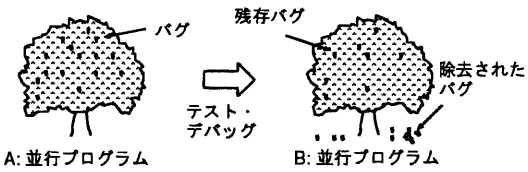


図 3: 従来のパラダイム

2.2.2 新しいパラダイム

超逐次プログラミングは，下記のステップから構成される (図 1)．この新しいパラダイムは，逐次化 (射影) と並行化 (復元) の組合せを特徴とする．

Step 1: モデル化とコード化
まず，設計対象を並行プロセスとしてモデル化し，並行プログラミング言語でコーディングする．もともと設計対象が並行性を持っている場合には，並行モデルは逐次モデルより自然であり，並行プロセスによるモデル化自体は難しくない．

Step 2: 逐次化 (射影)
並行プログラムを逐次化し，すべての非決定性を除去する．ここでは，元の並行プログラムのトポロジを残しながら，メタレベルの制御により逐次化を行なう．逐次化されたプログラムを「超逐次プログラム」と呼ぶ．保存されたトポロジは並行性を復元する段階で利用される．

Step 3: テストとデバッグ
逐次化されたプログラムに対しテストとデバッグを行なう．逐次化されているので，テスト・デバッグは容易である．

Step 4: 良い非決定性の導入
逐次化されたプログラム (超逐次プログラム) に対して，良い非決定性を明示的に 1 つ 1 つ導入する．各導入ごとに，その部分のテスト・デバッグを行う．

Step 5: 並行化 (復元)
最後に，プログラムの依存関係を解析し，無害な非決定性を自動的に検出し，並行性を可能な限り復元する．

図 2の例では，プロセス間の仮想的な優先度 ($P_1 > P_2$:P_1 の実行を P_2 より優先する) を導入することで並行プログラムを逐次化する．逐次化されたプログラムは 1 つの決定的な挙動 θ_1 しか持たない．それをテスト・デバッグすることは容易である．次に，良い非決定性として挙動 θ_2 を追加する．最後に，無害な非決定性 $\theta_4, \theta_5, \ldots$ を自動的に復元する．

新しいパラダイムは図 4のように表わせる．うっそうとした木 A を逐次化することによって，木の枝葉とそこに寄生していた虫が落ちてしまい，裸の幹 B だけが残る．この裸の木から残りの虫を除去するのは比較的容易である．しかし，この裸の木では元の姿 (機能) とはかけ離れている．そこで，バグのない枝 (良い非決定性) を 1 つ 1 つ調べながら明示的に追加し，木のだいたいの骨格 C を復元する．最後に，葉を自動的に復元してうっそうとした木 D を得る．木 D は木 A と比べると，形が若干いびつかもしれない．しかし，虫は駆除されている．

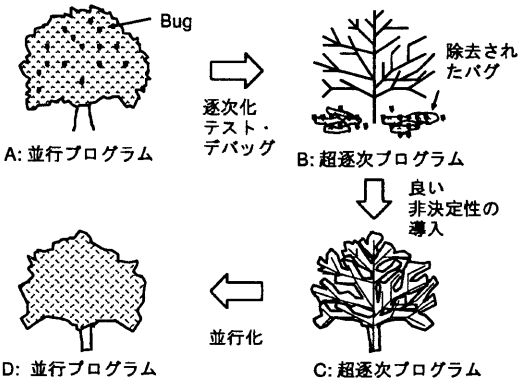


図 4: 新しいパラダイム

3 超逐次プログラミング

超逐次プログラミングの各ステップを説明する．

3.1 逐次化

メタレベル制御による逐次化としては下記の方法がある．

- **仮想的な優先度による逐次化:**
  並行に動作しうるプログラム単位 (プロセス，メソッド，基本ブロック) に仮想的な優先度を与えることにより，並行プログラムの挙動を決定的にする．
- **実行履歴による逐次化:**
  プログラムのある実行履歴を保存し，その履歴に基づいて並行プログラムの挙動を決定的に再現する．これも，1 つの逐次化の方法である．
- **シナリオによる逐次化:**
  実際の実行履歴のかわりに，プログラマが典型

的な実行順序をシナリオとして与える．シナリオに基づいて並行プログラムを実行するとき，挙動は決定的になる．

並行プログラムの逐次化は，挙動の再現性を確保するために，並行プログラムのテスト・デバッグではよく使われるテクニックである．典型的な例では，最初は1つのCPU上の疑似並列環境でテストを行ない，ある程度バグが枯れた段階でマルチCPU環境でテストを行なう，というものがある．しかし，従来のテスト・デバッグのための逐次化では，逐次化情報は挙動を再現するためだけに使われていた．超逐次プログラミングでは，逐次化情報を並行性の復元時にもデフォルトの実行順序として用いる点が大きな特徴である．

3.2 超逐次プログラム

オリジナルの並行プログラムのトポロジを保ちながら逐次化されたプログラムが超逐次プログラムである．超逐次プログラムは下記の3つの情報から構成される．

- セクション情報:

  プログラマは，オリジナルプログラムのソースコードをプログラムセクションと呼ぶ断片に分割する．テスト・デバッグにおいて，各セクションに属するプログラムコードは逐次的(決定的)に実行される．プログラムセクションは，基本ブロック(分岐や同期命令を含まないプログラムの処理単位)かもしれないし，メソッドのようにもっと大きな単位かもしれない．どの単位をプログラムセクションとするかは，プログラマが指定できる．

- プログラム依存グラフ:

  プログラム依存グラフは，プログラムセクション間の制御・データ依存関係を表現したグラフである．グラフの各ノードがプログラムセクションに対応する．

- 逐次化情報:

  逐次化情報は，逐次化後のプログラムセクション間の実行順序を表現したものである．この実行順序には，プログラム本来の「先天的な実行順序」と(本来は並行に動けるが)逐次化によって与えられた「後天的な実行順序」がある．後者をデフォルト実行順序と呼ぶ．デフォルト実行順序は，並行化の段階で明示的に除去されない限り保存される．

3.3 良い非決定性の導入

「良い非決定性の導入」は従来の並行プログラミングにはない，超逐次プログラミングに固有のステップである．良い非決定性の導入には下記の方法がある．

- デフォルト実行順序の除去:

  プログラマがGUIを用いてデフォルト実行順序を1つ1つ除去していくことで，良い非決定性を導入する[1]．図5は，デフォルト実行順序の除去過程を図示したものである．ここで，最初は全順序であった実行順序が，デフォルト実行順序を1つ1つ削除することによって，半順序化(並行化)していく．

- シナリオの追加:

  シナリオによる逐次化の場合，シナリオを追加することで良い非決定性の導入が実現できる．図2の例では，最初のシナリオθ_1にシナリオθ_2を追加することによって，良い非決定性が導入できる．このシナリオは，従来のプログラミングにおけるテストケースの作成に対応する．

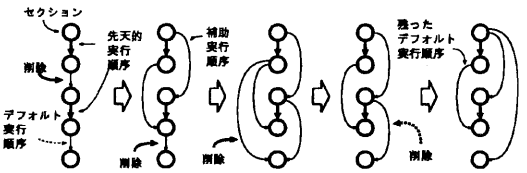


図 5: デフォルト実行順序の除去

3.4 並行化

逐次プログラムの並行化に関しては，スーパーコンピュータ用の最適化コンパイラ(具体的にはFORTRANプログラムの並行化など)の研究分野で多くの技術が開発されている[6]．超逐次プログラミングでは，これらの技術を活用し無害な非決定性の抽出および並行性の復元を行なう．

具体的には，良い非決定性の導入の終った超逐次プログラミングを以下の手順で並行化する．まず，セクション間の先行制約をプログラム依存グラフと逐次化情報から自動的に検出する．先行制約のないセクション同士は並行化可能であり，逐次化情報からそれらのセクション間のデフォルト実行順序をすべて削除する．最後に，残ったデフォルト実行順序

[1] ただし，先天的な実行順序は除去できない．

だけを保存するようにオリジナルの並行プログラムに同期命令を付加する[2]．最終的に得られた並行プログラムは，良い非決定性と無害な非決定性のみを持つ．

3.5 デフォルト逐次原理

超逐次プログラミングの特徴は，最初の逐次化で導入された実行順序が，良い非決定性の導入により明示的に除去されるか，または無害な非決定性として自動的に除去されない限り，最終的な並行プログラムにおいて保持される点である．この「明示的に非決定性を導入しなかった場合はデフォルトの実行順序を採用する」という発想は，AIにおけるフレーム問題の解として提案されたデフォルト論理と同じである．我々は，これをデフォルト逐次原理と呼ぶ．

我々がよく体験するバグに，あるプロセス P_1 が共有変数 x を初期化する前に，別のプロセス P_2 が変数 x を参照してしまう，というものがある．ところが，テスト段階ではたまたまプロセス P_1 が P_2 より先に実行されていたのでバグを発見できなったが，現地調整段階になってはじめてあるタイミングで P_2 が先に実行されバグが露見した，というケースは珍しくない．これは，プログラマは暗黙のうちに P_1 は P_2 より先に実行されるものであると思い込んでいたため，P_2 を先に実行するようなテストを行なわなかったからである．超逐次プログラミングでは，プログラマが P_2 が P_1 より先に実行するような場合 (非決定性) を明示的に導入しない限り，P_1 が実行されるまで P_2 の変数 x へのアクセスはデフォルトで抑止される．

4 いくつかの実施例

超逐次プログラミングは概念的なパラダイムの提案であり，様々な具体的実施方法が考えられる．我々は，ペトリネットを用いた超逐次プログラミング [3] とマルチタスク制御システムを対象としたシナリオベースの超逐次プログラミングを検討および試作している．これらを簡単に紹介する．

4.1 ペトリネットを用いた実施例

本実施例 [3] では，まず並行プログラムを高水準ペトリネットに変換し，下記のステップを実行し，最終的に得られたペトリネットから並行プログラムを逆生成する．

- 逐次化: ある優先度に基づきトランジションの決定的な発火順序 (デフォルト実行順序) を定める．このデフォルト実行順序を満たすように，ペトリネットに発火順序制御用のプレースとトランジションを追加する．
- テストとデバッグ: 逐次化されたペトリネットを実行し，テスト・デバッグを行なう．
- 良い非決定性の導入: プログラマが，発火順序制御用のプレースとトランジションを1つ1つ除去しながら，非決定的な挙動を増やしつつ，それらに対するテスト・デバッグを行なう．
- 並行化: 良い非決定性の導入が終了した段階で，ペトリネットの変換ルールを適用し，並行に動かしても影響のない部分を自動抽出し並行化する．結果的に，意味のない発火順序制御用のプレースとトランジションが除去できる．

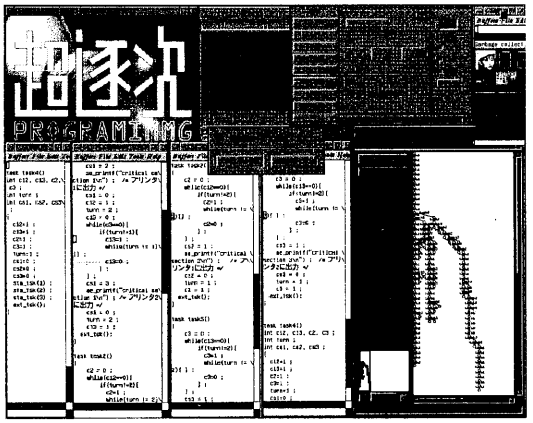
4.2 シナリオベースの実施例

複数のタスクが共有変数 (モニタ) を用いて同期通信を行なうプログラムを対象とする．

- 逐次化: タスクの実行順序をビジュアルに設定できるプログラミング環境 (図 6) を用いて，シナリオを作成する．
- テストとデバッグ: シナリオに基づいてプログラムを実行し，正しく動くか否かを確認し，バグがあればプログラムを修正する．
- 良い非決定性の導入: 重要なシナリオを追加し，各シナリオごとにテスト・デバッグを行なう．ここで，どのシナリオを追加するかよって，並行化における並行復元率が大きく異なる．
- 並行化: プログラマが明示的に与えたシナリオ群から，共有変数のアクセスパターンを抽出し，シナリオと結果が異なる可能性のあるアクセスパターンを許さないようにモニタのプログラムを改造する．ここで，プログラムの依存関係の解析により，実際のシナリオにはなかったアクセスパターンでも，無害な非決定性であれば復元される．

本手法に基づいて，10組のプログラマに対してデバッグの比較実験を行なった結果，従来手法では，与えられたデバッグ時間では平均 30 ～ 50 %のバグが残存したのに対し，本手法では 100 %除去できた．反面，並行性の復元に関しては従来手法の 70 %程度しか復元できなかった．

[2] プログラムに埋め込むかわりに，削除されなかったデフォルト実行順序を守るようにランタイムに制御する方式もある．



左下が対象となる並行プログラム (4プロセス)，右下のグラフが作成したシナリオを表す．

図 6: 超逐次プログラミング実験環境

5 関連研究

並行プログラムを逐次化してテスト・デバッグを行なうことは，よく用いられるノウハウである．このノウハウを体系化したものに [4] がある．また，逐次プログラムの並行化に関しては多くの文献がある [6]．特に，タスクレベルの並行性の自動抽出に関しては，Girkar と Polychronopoulos による提案 [5] がある．これらの技術は超逐次プログラミングにおける並行性の復元においても大いに活用できる．しかしながら，逐次化と並行化を組み合わせて，デフォルト逐次原理により高信頼な並行プログラムを開発する技術は，従来まったく存在しなかった．

データベースのトランザクションの並行制御 [7] は，超逐次プログラミングと技術的に共通点が多い．トランザクションの並行制御は，トランザクションが逐次的に処理された場合と結果が同じ範囲で並行処理を行ない，全体としての処理性能を高めるものである．ただし，トランザクションの並行制御は，性能向上が目的であり，並行プログラムの高信頼化が目的ではない．また，「良い非決定性の導入」という考え方もない．

シナリオに基づく超逐次プログランミングに関しては，メッセージシーケンスチャートによるシナリオから通信プログラムを自動生成する手法 [8] と関連がある．ただし，彼らの手法では全ての仕様をシナリオとして書く必要があるが，本手法におけるシナリオはオリジナルの並行プログラムに対する望ましい実行順序であり，シナリオ作成は容易である (従来のテストケース作成の延長である)．

6 おわりに

超逐次プログラミングの基本コンセプトと簡単な実施例を示した．超逐次プログラミングは，発電プラントなどの高い信頼性が要求される並行システムへ適用が有望である．今後の課題としては以下の項目がある．

- 並行性の復元率を向上させる要素技術の開発．この技術には，(1) 無害な非決定性を抽出するプログラム解析技術，(2) 並行性の復元に有効な良い非決定性を選択する技術，がある．
- 実時間制約がある並行プログラムに対する超逐次プログラミング手法の確立．現状では，最終的に復元された並行プログラムは，オリジナルより挙動が制限されるため，実時間制約を満たせない危険性がある．

謝辞:「超逐次プログラミング」は，ある社内プロジェクトの議論の成果である．プロジェクトメンバーである伊藤聡，植木克彦，大須賀昭彦，落合民哉，柿元満，澤島信介，塩谷英明，田原康之，永井保夫，半田恵一の各氏に感謝します．

参考文献

[1] McDowell, C.E. and Helmbold, D.P., Debugging Concurrent Programs, *ACM Computing Surveys*, Vol.21, No.4, 1989.

[2] Uchihira, N. and Honiden, S., Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *J. Systems Software*, **33**, 1996.

[3] Uchihira, N., Seki, T., Honiden, S., Hypersequential Programming, *Software Engineering for Parallel and Distributed Systems (Jelly, Gorton, Croll, eds.)*, Chapman & Hall, 1996.

[4] Bernstein, D. and So, K., Debugging Parallel Programs by Serialization, *United States Patent* No. 5048018, 1991.

[5] Girkar, M. and Polychronopoulos, C.D., Extracting Task-Level Parallelism *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.4, 1995.

[6] Zima, H. and Chapman, B., *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, 1990.

[7] Bernstein, F.A. and Goodman, N., Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, Vol.13, No.2, 1981.

[8] Ichikawa, H., Itoh, M., Shibasaki, M., Protocol-Oriented Service Specifications and Their Transformation into CCITT Specification and Description Language, *Trans. IECE Japan*, Vol.E-69, No.4, 1986.