プログラムの実行再現性を考慮した同期問題の解決法 *

1L−2

岡田 達彦
静岡大学 工学研究科†

太田 剛　水野 忠則
静岡大学 情報学部‡

1 はじめに

本稿では分散プログラムの同期問題 (デッドロック問題，リーダ／ライタ問題) の解決を補助する一手法について述べる．

同期問題を解決するためには，仕様に反する非決定的な挙動を取り除かなければならない．だが，この作業を計算機が自動的に行うことは難しい．なぜなら，非決定的な挙動の正誤の判定が，ユーザにしかできないからである．

ユーザは挙動の正誤の判定を行うために，ユーザ自身が意図していなかった挙動を再現させようとする．その結果，作成中のプログラムを何回か再実行させることになる．このとき，プログラムを最初から再実行させるか，またはプログラムの途中から再実行させるかは，ユーザ個人のもつ経験によって決まる．つまり，挙動の再現のために必要な命令を知っているかどうかが問題解決までの作業量に影響を与える．

この際に，計算機が挙動の再現に必要となる命令を自動的に求めることができれば，問題解決に到るまでの作業において，ユーザの経験に依存する部分を減らすことができる．本稿では，この必要となる命令群を半自動的に求める手法について記述する．まず2節において手法の導入にあたり必要な定義を示し，具体的な手法の説明は3節にて行う．

2 定義

2.1 対象とするプログラム

本稿で取り扱うプログラムは分散プログラムである．これは一時に複数個のプロセスをもつプログラムであり，プログラム中のプロセスは複数台の計算機にわたり分散して実行される．ただし，本稿においては，プログラム中のプロセス数は固定とする．

プログラム中のプロセスどうしの通信方式は，メッセージパッシング方式とする．任意のプロセスから送信されたメッセージは，受信先のプロセスのメッセージキューに入る．受信先のプロセスは，メッセージキューにメッセージが入るまでメッセージの受信処理の時点においてブロックする．

以降の記述において，単にプログラムとした場合には，上記の条件を満たすプログラムを指すものとする．

2.2 スペース・タイムダイアグラム

本手順の説明ではプログラムの一実行を記述するために，スペース・タイムダイアグラム [1] を用いる．以降の記述では，実行時点とは，スペース・タイムダイアグラム上の点を指すものとする．実行時点の種類の中には送信実行時点と受信実行時点とがある．送信実行時点においては他プロセスへのメッセージの送信が行われ，受信実行時点においてはメッセージキューからのメッセージの取り出し (メッセージの受理) が行われる．

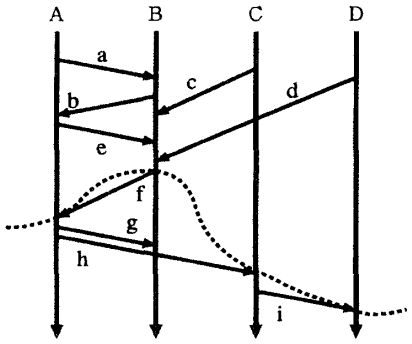

図 1: Check Point Frontier

2.3 *CheckPointFrontier*

本手順では *CheckPointFrontier*[2] を用いる．この導入にあたり必要となる定義を記述する．

- *maxavail*()

  $maxavail(r)$ は，受信実行時点 r にて受理できるすべてのメッセージをもつ集合である．ただし，r が属するプロセスが送信するメッセージを受理した後で生成されるメッセージは含まない．例えば，図1において，メッセージ d が受理される時点の *maxavail*() は {a,c,d} となり e は含まない．

- *CheckPoint*

  プロセス P の *CheckPoint* は，スペース・タイムダイアグラムにおいて送信実行時点 s より前の実行時点である．この s は $|maxavail(r')| = 1$ となる受信実行時点 r' の直後に位置する．また，プロセス開始地点も *CheckPoint* となる．

CheckPoint の導入により *CheckPointFrontier* は次のように定義できる．

- *CheckPointFrontier*

  1つのプロセス中の *CheckPoint* X 以降に送信されるすべてのメッセージを保留する．これにより保留されたメッセージを必要とするすべての他プロセスは，この送信に対応する受信実行時点でブロックする．このとき，*CheckPoint* X，およびブロックした受信実行時点にわたって引かれた線を *CheckPointFrontier* と呼ぶ．また，*CheckPoint*

---

*A Method to Reproduce a Execution Sequence in Distributed Programs

†Tatsuhiko Okada, Graduate School of Engineering, Shizuoka University

‡Tsuyoshi Ohta and Tadanori Mizuno, Faculty of Information, Shizuoka University

を含むプロセスを制御プロセスと呼ぶ．図1中の破線は，制御プロセスをBとした場合の *CheckPointFrontier* の例である．

2.4 *MinRange*

本手順の目的となる *MinRange* とその導入にあたり必要となる定義を記述する．

- $CP(i)$

  $CP(i)$ は制御プロセス中の *CheckPoint* を表す．i は制御プロセス内での時間順序に従ってつけられた通し番号である．これを用いると制御プロセス中の各 *CheckPoint* は $CP(1), CP(2), \cdots, CP(n)$ という表記になる．

- $CPF(x)$

  $x = CP(i)$ のとき，$CPF(x)$ は x によって定まる *CheckPointFrontier* を表す．

- $Range(A, B)$

  $A = CPF(a), B = CPF(b)\ (a < b)$ のとき，$Range(A, B)$ は，スペース・タイムダイアグラム上で $CPF(a)$，$CPF(b)$ によってはさまれる実行時点を要素としてもつ．

- $RangeNum(A, B)$

  $Range(A, B)$ に含まれる実行時点の個数を値として持つ．

  $RangeNum()$ は，$Range()$ どうしを実行時点の個数の大小によって比較することに用いる．例えば，$RangeNum(A, B) < RangeNum(C, D)$ のとき，$Range(A, B)$ は $Range(C, D)$ よりも少ないと表す．

- *MinRange*

  MinRange は次の2つの条件を満たす $Range()$ である．

  (1) ユーザがプログラム中のある一つの挙動を再現しようとしたとき，その挙動の発生に関与した実行時点をすべて含む．すなわち，*MinRange* に含まれている実行時点だけで，目的の挙動を再現できる．

  (2) 実行時点の個数が極小である．

3 手順

本節では2節で定義した *MinRange* を求める手順を説明する．定義からわかるように *MinRange* は，挙動の再現にあたり必要となる実行時点をすべて含む極小の実行時点集合である．

手順の説明に入る前に，計算機側とユーザ側に次の仮定をおく．

- 計算機側: 次の事柄を行い，結果をユーザに提供できると仮定する．
  - プログラムの一実行を反映したスペース・タイムダイアグラムを作成する．
  - プローブ効果なしでプログラムの実行履歴，プロセス間のメッセージ送受信の履歴を記録する．
  - $Range()$ に含まれている実行時点の命令をプログラムのソースから取り出し，単独で実行させる．
- ユーザ側: プログラム，及び $Range()$ の実行時に，再現を試みている挙動が起こっているかどうかが分かる．

上記の仮定のもとで，*MinRange* を求める手順をC風のプログラムで記述する．対象となるプログラムを P とし，P の制御プロセスは *CheckPoint* を n 個もつとする．また，ユーザによって，P のもつプロセス群から一つのプロセスが制御プロセスとして選ばれているものとする．

```
a = 1; b = n; a' = (a+b)/2; b' = n;
while (|a-a'| != 0) {
     if ( check(Range(a',b)) == 1 ) {
          a = a';
          a' = (a'+b')/2;
     } else {
          b' = a';
          a' = (a+a')/2;
     }
}
UpperLim = a;
a = 1; b = n; a' = 1; b' = (a+b)/2;
while (|b-b'| != 0) {
     if ( check(Range(a,b')) == 1 ) {
          b = b';
          b' = (a'+b')/2;
     } else {
          a' = b';
          b' = (b+b')/2;
     }
}
BottomLim = b;
MinRange = Range(UpperLim, BottomLim);
```

手順中の関数 check() は，引数として与えられた $Range()$ が *MinRange* の条件 (1) を満たしているかどうかを判定する関数である．条件を満たしていれば1を，満たしていなければ0を返す．

4 おわりに

本稿で説明した *MinRange* は，同期問題の発生が確認された後で必要となる，プログラム中のある一つの挙動を再現させるときに役立つ．その利点として，(1)*MinRange* に含まれる実行時点を調べることで挙動の再現に必要な命令がわかる，(2) 極小という性質から，ユーザ側の調査範囲の削減に役立つ，(3)*MinRange* に含まれる実行時点の命令群をプログラムのソースから取り出して単独で実行しても，元のプログラムの挙動と同じ実行結果が得られる，ということが挙げられる．

参考文献

[1] Lamport,L.: "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. ACM, Vol.21, No.7, pp.558–pp.565 (1978).

[2] Damodaran-Kamal,S.K. and Francioni,J.M.: "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs," ACM SIGPLAN NOTICES, Vol.28, No.12, pp.118–128 (1993).