マルチメディア通信と分散処理 77－7
（1996．7．12）

分散プログラムにおけるデッドロック原因究明の定式化

岡田 達彦，太田 剛，水野 忠則
okada@cs.inf.shizuoka.ac.jp
静岡大学情報学部
〒432　浜松市城北三丁目5番1号

本稿では，分散プログラムにおけるデッドロック問題を解決する一手順について述べる．問題の原因を究明するために，本方式では，動的スライスと時間ベクトルを用いる．動的スライスは，プログラムの実行系列から抽出された，デバッグに必要な実行時点集合である．時間ベクトルは，動的スライス中の実行時点を半時間順序に従って整列するために用いる．この整列によって，デッドロックを引き起こす原因となる，時間順序の不整合を起こしている送受信文列を発見できる．

Towards a Formalization of a Fault Localization Scheme for Deadlocks in Distributed Programs

Tatsuhiko Okada, Tsuyoshi Ohta and Tadanori Mizuno
Faculty of Information, Shizuoka University
3-5-1, Johoku, Hamamatsu, Shizuoka, 432 Japan

In this paper, we describe a scheme applicable to deadlock problems in distributed programs. This scheme uses dynamic slicing and vector clock. Dynamic slice is an ordered set of events chosen from an execution instance of a program so as to be useful for debugging. Vector clock is used to make events sorted in partial order. The sorted events allow us to detect a contradictory sequence of receive/send events which would cause a deadlock.

1 はじめに

分散処理環境下で稼働する分散プログラムの開発には，従来の逐次型プログラムにはない難しさが存在する．それは，プログラムの挙動の非決定性，プローブ効果，絶対時間の欠如等に起因する [1, 2]．特に分散プログラムに固有のデッドロック問題がデバッグを困難にする．これらの要因のため，分散プログラムのデバッグは試行錯誤で進められることが多い．

逐次型プログラムのデバッグ技術の1つとして，プログラムスライシング技術がある．プログラムスライシング技術は，プログラム中の文の間に存在する依存関係をもとに，プログラム中のある特定の文に影響を与える文を抽出する技術である．この技術を用いることにより，プログラムテキスト上でのバグの探索範囲を狭めることができる．現在，プログラムスライシング技術は，プログラムのデバッグに限らず，テスト・保守・プログラム理解へと広範囲に応用されている [3]．

プログラムスライシング技術を逐次型プログラムのデバッグに適用した研究には，Weiser[4] がある．分散プログラムに適用した研究には，Cheng[5]，Taylor[6] がある．Cheng は並行プログラムのスライスを定義し，Taylor はデータフロー分析によりシングルプロセスと並行プロセスの両方に対してプログラムの異常を発見できることを示している．文献 [5, 6] が示すように，プログラムスライシング技術を適用することにより，分散プログラムのデバッグに対しても，逐次型プログラムと同じようにデバッグに必要な情報を削減できる．

分散プログラムにプログラムスライシング技術を適用して，文の記述の誤りや変数値の誤りを修正できたとしても，分散プログラムのデッドロック問題は解決できない．なぜなら，個々のプロセスが正しく動作したとしても，複数のプロセスが同時に動作したときの相互作用がデッドロックを引き起こすからである．このため，デッドロック問題を解決するためには，ユーザが試行錯誤でテストを繰り返して，デッドロックに至る実行過程を追及することになり，多大な労力が必要となる．

この問題を解決するために，本稿では，動的スライス [7, 8] を用いた，デッドロック問題の解決方法を定式化する．本方式を適用することによって，プログラムの実行系列中からデッドロックを引き起こした実行時点を半自動的に求めることができ，デバッグ作業に対するユーザ側の負担を減らすことができる．

2 プログラムのモデル

対象とするプログラム言語は，構造型プログラミング言語に最低限必要な文に，プロセス定義，送信文，受信文を追加した手続き型言語とする．すなわち，この言語は，空文，代入文，if 文，if-else 文，while 文，入力文，出力文，送信文，受信文，変数宣言，プロセス定義から構成される．この文法は，プロセスの生成 (fork)，結合 (join) を行う文を含まない．また，関数，手続きの定義はない．変数は個々のプロセス内で有効なスカラ型に限り，共有変数は使用できない．プロセス間の通信方式は，同期式メッセージパッシング方式とする．以後，単にプログラムと記述した場合には，この言語で記述されたプログラムを指すものとする．

3 デッドロックの原因究明

本方式は，プログラムの実行時にデッドロックが検出された後，プログラムの実行系列の中からデッドロックを引き起こした実行時点を発見するために用いられる．本方式を用いた，デッドロックの原因を究明するための手順は，次のようになる．

1. スライスの計算
2. 時間ベクトルの計算
3. 原因の究明

この一連の手順は，現在起こっているデッドロックを引き起こしている原因が除去されるまで，複数回実行される．1では，4節で説明するプログラムスライシング技術により，各プロセスごとにスライスを計算する．2では，5節で説明する計算法により，スライスに含まれている実行時点の時間ベクトル [9] を計算する．6節では，プログラム中の時間順序の不整合を発見することが，デッドロックの原因を解決する手掛りになることを示す．そして，1 ～ 3の各手順の詳細を述べる．また，各手順は，以下の仮定のもとで，実行される．

1. ユーザが (a) プログラムの制御移行の正誤と (b) プログラム中の送受信文の組み合わせの正誤を判断できる．
2. プログラムの仕様は正しい．
   本方式は，プログラムが仕様によってデッドロックを起こしている場合には，原因を究明することができない．例えば，'哲学者の会食' にて生じるデッドロック問題は，本方式では解決できない．
3. プログラムがデッドロックを起こすまでの実行履歴を常に参照できる．

4 分散プログラムの動的スライス

プログラムの実行系列から動的スライス [7, 8] を計算する方法を示す．

4.1 定義

最初に種々の定義を示す．これらは逐次型プログラム上での文献 [8] の定義の自然な拡張となっている．また，本節以降，プログラム P 内のプロセス集合を $\{P_1, P_2, \cdots, P_i, \cdots, P_n\}$ と表記する．

- プロセス実行系列の定義
  プロセス P_i の実行系列とは，P_i において実行された命令の列である．そして，実行時点 p とは，p 番目に命令の実行が行われた時点を示す．P_i の実行時点を $\{e_1^{P_i}, e_2^{P_i}, \cdots, e_x^{P_i}, \cdots, e_m^{P_i}\}$ と表記する．

- Ins の定義
  プロセス P_i $(1 \leq i \leq n)$ に含まれる実行時点 p において実行された命令を $Ins(p)$ と定義する．

- Def の定義
  実行時点 p における命令 $Ins(p)$ の実行により定義された変数の集合．

- Use の定義
  実行時点 p における命令 $Ins(p)$ の実行により使用された変数の集合．

- $LastDef$ の定義
  $p = LastDef(q, w)$ とは，実行時点 q において変数 w が保持している値は，実行時点 p で定義されたことを表す．ただし，$p < q$ で，p と q は，同一プロセス内に存在する．

- $ConDep$ の定義
  $ConDep(s, t)$ とは，命令 t が命令 s に，プログラムテキスト上で制御依存していることを表す．すなわち，命令 s が分岐命令 (または，ループ命令) であり，命令 t がその分岐文内 (ループ文内) に直接含まれていることを示す．命令 t が分岐文 s 内に直接含まれているとは，分岐文 s 内の他の分岐文やループ文に命令 t が含まれていないことを意味する．

- $CtlExec$ の定義
  $CtlExec(t) = \{$命令$s \mid ConDep(s, t)\}$．ただし，命令 t がループ命令の場合には命令 t も $CtlExec(t)$ に含める．

- cor の定義
  $p = cor(q)$ は，送信文実行時点 p と受信文実行時点 q とが一対一に対応付けられていることを表している．$p \in P_i$, $q \in P_j$ $(i \neq j)$に対して，$p = cor(q)$ が成立するのは次の場合である．

  1. プログラム実行中，pとqとの間で送受信が行われた．
  2. ユーザがpとqとを対応付けた．

スライス基準を次のように定義する．

- スライス基準: $C = (x, i, r, V)$
  x: プログラム P に与えた入力．
  i: プロセス識別子．
  r: プログラム P に入力 x を与えたときの，プロセス P_i の実行系列上の実行時点．
  V: プロセス P_i 内の変数の部分集合．

4.2 依存関係

動的スライスの計算に必要な依存関係は，次の2つである．

- データ依存関係: $DU(p, q)$
  ある変数 $w \in Use(q)$ が存在して，$p = LastDef(q, w)$ の場合．あるいは，$p = cor(q)$ の場合に成立．

- 制御依存関係: $TC(p, q)$
  $p = max\{$実行時点$i \mid i < q$，かつ $Ins(i) \in CtlExec(Ins(q))\}$ の場合に成立．

4.3 スライス (DS) の計算

プロセス $\{P_1, \cdots, P_i, \cdots, P_n\}$ の実行系列から，スライス基準 $C = (x, i, r, V)$ で動的スライス $DS(P)$ を計算するためには，次のようにする．

(1) $A^0 = \{q \mid$ ある変数$v \in V$に対して $q = LastDef(r, v) \vee q = cor(r) \vee TC(q, r)\}$.

(2) $n \geq 1$ に対して
$A^n = \{p \mid$ ある実行時点$q \in A^{n-1}$が存在して，$Rel(p, q)\}$．ただし，$Rel = DU \cup TC$.

(3) $DS(P) = \{Ins(p) \mid p \in \bigcup_{n=0}^{\infty} A^n \cup \{r\}\}$.

5 時間ベクトル

プログラムの実行系列に含まれる実行時点の時間ベクトル [9] を計算する方法を示す．

$e_x^{P_i}$ のもつ時間ベクトル $T_x^{P_i}$ を計算するために，関数 $tcalc$ を用いる．

$tcalc(e_x^{P_i}) = T_x^{P_i} = \{t_{P_1}^{P_i}, t_{P_2}^{P_i}, \cdots, t_{P_n}^{P_i}\}$.

ここでの t は，P_i からみた各プロセスのローカル時間を表す整数値である．例えば，$t_{P_x}^{P_i}$ は，プロセス P_i からみたプロセス P_x のローカル時間を値としてもつ．

$tcalc$ は次の手順により時間ベクトルを計算する．

(1) プロセス P_i の最初の実行時点 $e_0^{P_i}$ の時間ベクトルは，$T_0^{P_i} = \{0, 0, \cdots, 0\}$ である．

(2) $e_x^{P_i}$ の時間ベクトルが $T_x^{P_i} = \{t_{P_1}^{P_i}, t_{P_2}^{P_i}, \cdots, t_{P_i}^{P_i}, \cdots, t_{P_n}^{P_i}\}$ であるとき，

  (a) $e_{x+1}^{P_i}$ が，代入文，入力文，出力文，送信文のいずれかの実行時点ならば，
  $T_{x+1}^{P_i} = \{t_{P_1}^{P_i}, t_{P_2}^{P_i}, \cdots, t_{P_i}^{P_i} + 1, \cdots, t_{P_n}^{P_i}\}$.

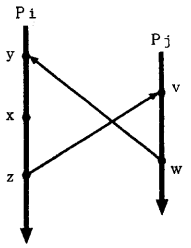

図 1: 時間ベクトルが計算できない状態

(b) $e^{P_i}_{x+1}$ が受信文実行時点で, $cor(e^{P_i}_{x+1}) = e^{P_j}_{y}$ であるならば,
$tcalc(e^{P_j}_{y}) = T^{P_j}_{y} = \{t^{P_j}_{P_1}, t^{P_j}_{P_2}, \cdots, t^{P_j}_{P_i}, \cdots, t^{P_j}_{P_n}\}$,
$T^{P_i}_{x+1} = \{max(t^{P_i}_{P_1}, t^{P_j}_{P_1}), max(t^{P_i}_{P_2}, t^{P_j}_{P_2}), \cdots,$
$max(t^{P_i}_{P_i}+1, t^{P_j}_{P_i}), \cdots, \max(t^{P_i}_{P_n}, t^{P_j}_{P_n})\}$.

時間ベクトルを計算できない場合

デッドロックの起きない実行系列では，上記の方法ですべての実行時点の時間ベクトルを求めることができる．だが，デッドロックを起こした実行系列では，時間ベクトルを計算できない場合が存在する．それは，次のような場合である．

プロセス P_i に含まれる実行時点 $e^{P_i}_{x}$ の時間ベクトル $T^{P_i}_{x}$ の計算を考える．この計算で，呼び出された $tcalc$ が，プロセス $P_{j(\neq i)}$ の実行時点 $e^{P_j}_{w}$ の時間ベクトルを求めようとしたとき，$e^{P_j}_{v(<w)}$ があり，かつ，$e^{P_i}_{z(>x)} = cor(e^{P_j}_{v})$ であれば，時間ベクトル $T^{P_i}_{x}$ は求めることはできない．図 1は，その様子を表している．

6 デッドロック解決への手掛り

本節では，プログラム中に存在するサイクルを見つけることが，デッドロック問題の解決に役立つことを示す．

最初に必要な定義を述べる．

6.1 サイクルの定義

＜'→' の定義＞

'→' は，次の 3 つの状態のいずれかを満たす，2 つの実行時点間の関係である．

(i) a と b が同じプロセス上の実行時点で，a が b よりも前に実行された場合．
$a \rightarrow b$, $(\exists a, \exists b \in e^{P_i})$.

(ii) あるプロセス上の実行時点 a から，もう 1 つの別のプロセス上の実行時点 b へと実際に値の受け渡しがあった場合．
$a \rightarrow b$, $(\exists a \in e^{P_i}, \exists b \in e^{P_{j\neq i}})$.

(iii) あるプロセス上の実行時点 a から，もう 1 つの別のプロセス上の実行時点 b へと値の受け渡しがあったと仮定した場合．
$a \rightarrow b$, $(\exists a \in e^{P_i}, \exists b \in e^{P_{j\neq i}})$.

また，以上の 3 つの場合において，$a \rightarrow b$ かつ，$b \rightarrow c$ ならば，$a \rightarrow c$ である．

＜サイクルの定義＞

プログラム中のある 2 つのプロセス (P_i, P_j) がサイクルを形成しているとは，次の場合のことをいう．

$\exists a, \exists b \in e^{P_i}$, $\exists s, \exists t \in e^{P_j}$ が，
$a \rightarrow b$, $b \rightarrow s$, $s \rightarrow t$, $t \rightarrow a$ という関係を満たす．

図 2は，この様子を表している．また，上式は 2 個のプロセスがサイクルを形成する場合だが，n 個のプロセスがサイクルを形成する場合も同様の式で表される (→ の定義より)．図 3は，n 個のプロセスがサイクルを形成している様子を表している．

6.2 サイクルの発見

2節の言語には，共有リソースの定義は含まれない．これにより，このプログラムで起こるデッドロックの原因は，送信文の記述漏れとサイクルの 2 つに限られる．

送信文の記述漏れとは，プログラム中のある受信文に対応するべき送信文が漏れていることを指す．これに相当する受信文は，送信されることのないメッセージを待ち続けることになり，デッドロックを引き起こすことがある．この誤りは，実行時点間の依存関係を調査することで，容易に修正できる (6.3.1節参照)．デッドロックの原因とはならないが，受信文の記述が漏れている場合も，同じ手順により修正できる．この誤りを修正してもプログラムがデッドロックを起こすならば，サイクルが原因である．

＜定理 1 ＞

2節の言語で記述されたプログラムが送信文の記述漏れ以外の原因でデッドロックを起こしたときには，サイクルが存在する．

(証明)

2節の言語で記述されたプログラムが，サイクルと送信文の記述漏れ以外の原因によりデッドロックを起こしたとする．デッドロックを起こす原因となるのは，送信文の記述漏れがないと仮定すると，(1) サイクル，(2) 共有リソース取得時の複数プロセス間での競合の 2 つである [10]．2 節の言語には，共有リソースの定義がないため，サイクルと送信文の記述漏れ以外の原因によりデッドロックを起こしたことに矛盾する．

(証明終わり)

＜定理 1 ＞から，サイクルを構成するプログラム中のプロセス，及び，実行時点を調査し，サイクルを解除

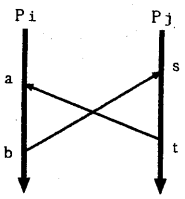

図 2: サイクルの例 1

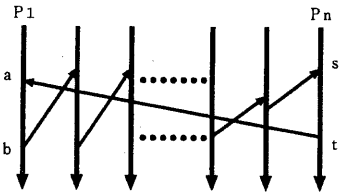

図 3: サイクルの例 2

することによって，デッドロック問題を解決できることが分かる．サイクルを構成する実行時点を探すためには，時間ベクトルを計算すればよい．時間ベクトルが計算できない実行時点が，サイクルを構成する．

＜定理 2＞

時間ベクトルが計算できないプログラム中の実行時点の集合は，サイクルを構成する．また，サイクルを含むプログラムには，時間ベクトルが計算できない実行時点が存在する．

(証明)

ある受信文実行時点 $a \in P_i$ の時間ベクトルを求めるとき，$tcalc(a)$ が実行される．
$cor(a) = t \in P_j$ であれば，t の実行時点を求めるために再帰的に $tcalc(t)$ が実行される．このときの t と a の関係は $(1)t \rightarrow a$ である．

再帰的に呼び出された $tcalc(t)$ は，t の時間ベクトルを計算するために，P_j の (t より前の) 実行時点の時間ベクトルを求める．このとき，受信文実行時点 $s \in P^{P_j}$ $(s < t)$ があり $cor(s) = b \in P_i$ ならば，$tcalc(b)$ が実行される．これらの関係は，それぞれ $(2)s \rightarrow t$，$(3)b \rightarrow s$ である．

3 節での時間ベクトルが計算できない状態というのは，$b > a$ となるときである．このときの 2 者の関係は $(4)a \rightarrow b$ である．(1),(2),(3),(4) の示す式は，サイクルの式となる．

$a, b \in P_i$，$s, t \in P_j$ に対して，
$a \rightarrow b$，$b \rightarrow s$，$s \rightarrow t$，$t \rightarrow a$.

また，上記のサイクルの式は，
$a < b$，$cor(s) = b$，$s < t$，$cor(a) = t$ と表すことができる．これは，5節の時間ベクトルを計算できない場合と一致する．

(証明終わり)

6.3 デッドロック原因究明の手順

デッドロック原因究明の手順を構成する 3 つのフェーズ (1. スライスの計算，2. 時間ベクトルの計算，3. 原因の究明) の詳細を述べる．

6.3.1 スライスの計算

3 番目のフェーズ: 原因の究明は，ユーザとの対話により進行する．このため，ユーザに不必要な情報を提示することがないように，システムはプログラムがデッドロックを起こしたときの実行履歴より，4節で説明した方法に従ってスライスを計算する．この計算を行うことで，実行系列の中からデッドロックの原因究明に必要な実行時点だけをスライスとして抽出できる．また，スライスを計算するときに参照される実行時点間の依存関係は，ユーザとの対話のときにも参照される．

スライスの計算は，実行履歴中のデッドロックを起こしたプロセスすべてを対象として行う．このときのスライス基準は，(1) プログラムに与えられた入力，(2) デッドロックを起こしている受信文を含むプロセス識別子，(3) デッドロックを起こしている受信文，そして，(4) その受信文に含まれている変数とし，各プロセスごとに個別に設定する．

スライスを求めるためには，プログラム中の送受信文は，一対一に対応していなければならない．プログラム実行時に実際に通信が行われた場合には，対応付けは容易に行われる (4.1節,*cor* の定義 -1)．だが，デッドロックを起こしている受信文実行時点には，対応する送信文実行時点はない．この場合，システムはユーザに問い合わせ，ユーザの指示により送受信文間の対応付けを行う (4.1節,*cor* の定義 -2)．このとき，もし，対応する送信文がない受信文が発見されたならば，プログラムに送信文の記述漏れがある．

プログラム P の実行時にデッドロックを起こしたプロセスが $\{P_a, P_b, \cdots, P_y, P_z\}$ であり，各プロセスに対するスライスが
$\{DS(P_a), DS(P_b), \cdots, DS(P_y), DS(P_z)\}$ であったとき，スライス集合:DSS は次式で表される．

スライス集合 : DSS
$= DS(P_a) \cup DS(P_b) \cup \cdots \cup DS(P_y) \cup DS(P_z)$.

このフェーズにおいて送信文の記述漏れが発見されたならば，必要な送信文を補うことによって，現在起こっているデッドロック問題を解決できることがある．システムは，ユーザから与えられた指示に従って，送信文を補う．記述漏れがない場合，または，記述漏れを補ってもデッドロックが起きる場合，次のフェーズ：時間ベク

トルの計算に移行する．

6.3.2 時間ベクトルの計算

プログラム中のサイクルを発見するために，スライス集合:DSS に含まれる実行時点すべてに対して，時間ベクトルを計算する．計算法は5節の記述に基づく．時間ベクトルを計算できない実行時点が見つかった時点で，次フェーズ：サイクルの解除に移行する．

6.3.3 原因の究明

＜定理2＞により，時間ベクトルを計算できない実行時点はサイクルを構成する．このサイクルを解除することによって，デッドロック問題を解決する．サイクルの解除は，ユーザとの対話によって得られた情報に基づいて行われる．

サイクルの原因には，次の2つが考えられる．

1. プログラムの命令の実行順序の誤り．
2. プログラムの制御移行の誤り．

前者の場合は簡単に解決する．例えば，図2の場合では，実行時点 a と実行時点 b(または，s と t) における命令の実行順序を入れ換えればよい．システムは，サイクルを構成している実行時点をユーザに示し，ユーザから入れ換えが可能な実行時点の情報を得る．

後者の場合には，プログラムの制御移行の誤りにより，ユーザが意図するものと異なる送受信文どうしが，実行時に対応付けられている．システムは，この送受信文の対応付けに至るまでの実行系列，系列上の時間ベクトル値，及び，送受信文の対応状況をユーザに示す．ユーザは，システムから与えられた情報をもとに，制御移行の誤りを生じさせる箇所を究明していく．

ここでシステムが表示する情報は，ユーザが理解しやすいように整理されていなくてはならない．このために，システムは必要な実行時点を半時間順序に従って整列表示し，同時に実行時点間の依存関係を表示する．この整列表示の際には，前のフェーズで計算された実行時点の時間ベクトル，及び，スライスの計算時に参照された実行時点間の依存関係が用いられる．

以上3つのフェーズの終了後，プログラムを再実行する．これによって新たなサイクルが見つかったならば，再びスライスを計算し，同様の手順でサイクルを解除する．

7 おわりに

3節の3つの仮定のもとで，プログラムがデッドロックを起こしたときの原因が，サイクルにあることを示し，プログラム中のサイクルを解除する方法について述べた．本方式を適用することによって，プログラム中のサイクルを構成する送受信文の組み合わせを半自動的に求めることができる．また，プログラム中の実行時点間の依存関係，及び，実行時点の時間ベクトルを表示できる．これは，ユーザが複雑なプロセス間通信の様子を理解するのに役立つ．

ただし，3節の仮定2が示すように，プログラムの仕様によりプログラムがデッドロックを起こしている場合には，本方式は原因を究明することはできない．この場合，ユーザによるプログラムの仕様の再検討が必要となる．

今後，実用的なプログラムに対して本方式を適用し，評価を行う予定である．

参考文献

[1] McDowell,C.E. and Helmbold,D.P.: "Debugging Concurrent Programs," ACM Computing Surveys, Vol.21, No.4, pp.593–622 (1989).

[2] 山田: "並列処理におけるプログラムデバッグ", 情報処理, Vol.34, No.9, pp.1170–1178 (1993).

[3] 下村: "Program Slicing 技術とテスト, デバッグ, 保守への応用", 情報処理, Vol.33, No.9, pp.1078–1086 (1992).

[4] Weiser,M.: "Program Slicing," IEEE Trans. on Software Engineering, Vol.SE–10, No.4, pp.352–357 (1984).

[5] Jingde Cheng: "Slicing Concurrent Programs," in "Automated and Algorithmic Debugging" (Lecture Notes in Computer Science 749), Springer–Verlag, pp.223–240 (1993).

[6] Taylor,R.N. and Osterweil,L.J.: "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Trans. on Software Engineering, Vol.SE–6, No.3, pp.265–278 (1980).

[7] Korel,B. and Laski,J.: "Dynamic Program Slicing," Information Processing Letters, Vol.29, No.10, pp.155–163 (1988).

[8] 下村: "プログラムスライシング技術と応用", 共立出版 (1995).

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

[10] 亀田, 山下: "分散アルゴリズム", 近代科学社 (1994).

[11] 太田, 渡辺, 水野: "プログラムスライシングの分散プログラムへの適用", 情報研報, マルチメディア通信と分散処理,65–8,pp.43–pp.48 (1994).

[12] 岡田, 太田, 水野: "プログラムスライスを用いた分散プログラムのデッドロック原因究明方式", 情処マルチメディア通信と分散処理ワークショップ論文集, Vol.95, No.2, pp.9–pp.15 (1995).