5 並列プログラムのテスト

片山徹郎[*1]・高橋寿一[*2]

[*1] 宮崎大学 工学部 情報システム工学科　[*2] ソニー(株)

近年，エンジニアの言う「タイミング依存のバグ」が現場では多くなっているのではないか．昨今の大きなシステム障害を見ると，「あるまれなタイミングと，あるまれなタイミングが同時に起こると障害が発生する」という記事をよく見かける．また，その障害に対する記者会見では「単独製品としては起こり得ない」，もしくは，「弊社製品単独の機能としては問題ないのだが，他社製品と統合した場合に問題が発生する」という言葉をよく聞く．

現代のソフトウェアやシステムは，単独機能として大きな問題を起こすことは少なくなってきている．それは各社の品質保証システムや部門が成熟した成果であろう．しかし，そのソフトウェアやシステムの堅牢性を維持するために多重化したり，あるいは，高速な処理性能を求めるために並列化することによる起こるバグについては，逆に増える傾向がある．

本稿では，前述したような並列に処理されるプログラムやシステムについて，テストの難しさの原因とテスト手法について述べる．また，並列プログラムの特性と誤りの分類について述べ，その中から特に，デッドロックの検出手法と競合状態のテスト手法について述べる．

並列プログラムを取り巻く背景

並列プログラムについては，テストのみならず，開発やプログラミングも含めて，古くて新しい課題であると言える．1980年代から90年代にかけて，「今後，コンピュータシステムの処理性能向上のために，並列分散処理の時代がくる」と言われていた．現実には，ハードウェアの発展，特にCPUの動作周波数の向上が著しかったため，並列処理が強く望まれることはなかった．また，この時期は，Webとネットワークの技術が発展するとともに，その普及が一段と進み，技術者や研究者はこれらの分野に力を注ぐことが多かったために，並列処理のための言語や枠組みなどについて，あまり議論が進展しなかった．

このため，並列処理が持つ本質的な難しさやその原因については，1980年代からすでに指摘されており，現在もそれほど変わっていないと言える．一方，ここにきて，マルチコアやマルチスレッドに代表されるように，並列プログラムが望まれる背景が整ってきた．すなわち，マルチコアやマルチスレッドを用いることによって，ソフトウェア処理の並列性を高め，システム全体の処理性能を上げようとする考え方である．

実際，各プロセッサメーカは，CPUの動作周波数をこれ以上飛躍的に向上させることは困難な現状があるため，1つのCPUパッケージ内に複数のCPUコアを封入したマルチコアに向かっている．しかし，ソフトウェアが複数のCPUコアを効率的に利用できなければ，意味がなくなる可能性がある．このため，並列コンピューティングに対応したプログラミングが必要となり，ソフトウェアの開発は難しくなる．開発が難しくなる理由の1つに，複数のプロセスやスレッドの処理が共有資源にアクセスする場合に，ロックなどを使った排他制御（相互排除とも言う）が必要なことが挙げられる．排他制御とは，複数のプロセスが利用できる共有資源に対し，複数のプロセスからの同時アクセスにより競合が発生する場合に，あるプロセスに資源を独占的に利用させている間は，他のプロセスが利用できないようにすることによって，データの一貫性（整合性）を保つ処理のことである．

排他制御によりロックされた共有資源に，複数のプロセスからアクセス要求が出された場合に，お互いに資源が解放されるのを待ち続けるという状況が発生する．このような状態に至ると，どのプロセスも処理を進めるための必要な資源が確保できないため，停止状態となる．この状況は，決して起こらない特定の事象を待っていることから，デッドロックと呼ぶ．多数のロックを使う処理は，デッドロックに陥りやすい．

また一方で，ロックを適切に記述していない場合，データの更新タイミングを考慮に入れていないプログラムモデルでは，排他制御がうまく働かなくなる可能性がある．この場合，システム内のデータの一貫性が壊れる事態が生じる．並列プログラムを正しく動作するように記述するためには，並列処理の難しさを十分に理解し，起こり得る誤りについて熟知しておく必要がある．

5 並列プログラムのテスト

本稿では，並列プログラムのテストの難しさの原因とテスト手法について解説する．以降の章では，並列プログラムの特性とプログラムで発生する誤りの分類について述べる．その中から特に，生存性の破壊誤りであるデッドロックの検出手法，および，安全性の破壊誤りと関連する競合状態のテスト手法について述べる．

並列プログラムの特性と誤りの分類

並列プログラムは，逐次的に実行される複数のプログラム単位が通信やデータ同期といった相互作用を行いながら，処理を進めるプログラムである．プログラム単位は，プログラミング言語やプログラムの種類によって，プロセスやタスク，ジョブと，いろいろな呼び方が存在する．本稿では，静的な概念としてはプログラム単位と，動的な概念としてはプロセスと，呼ぶこととする．

並列プログラムは，逐次プログラムと比較すると，並列処理のための記述が追加され，順次行う処理を同時に進めることが可能となるので，全体としての処理時間を短縮できる可能性がある．一方で，並列に動作するプログラム単位を適切に制御するための機構が必要である．制御が適切でないと，逐次プログラムよりも処理が遅くなる可能性も生じる．さらに個々のプロセスは並列に動作する他のプロセスとの間での，通信や同期といった相互作用があり，その相互作用が適切に行われなければならない．相互作用が適切でない場合には，プログラムそのものが動作しなくなるデッドロックの状態に陥ったり，データの一貫性が壊れ計算結果が矛盾する，という状況も起こり得る[1]．

並列プログラムの誤りを分類すると，

- 計算誤り
- 通信誤り
- 同期誤り

がある[2]．

このうち，計算誤りとは，逐次プログラムにおいても発生する誤りであり，逐次的に実行されるプロセス内で発生する誤りである．通信誤りは，2つのプロセス間でのデータの受け渡しに伴う誤りである．逐次プログラムでも，手続きや関数の呼び出し時に発生する．同期誤りは，プロセスの実行順序が不適切であることによって発生する誤りであり，並列プログラム特有の誤りであると言える．

さらに，この同期誤りは，

- 生存性の破壊誤り
- 安全性の破壊誤り
- 公平性の破壊誤り

に分類できる．

生存性の破壊誤りは，プロセス間での同期の失敗によって，デッドロックが発生する誤りである．たとえば，よく知られた問題として Dijkstra が提案した「哲学者の食事問題」がある．中央にスパゲティが盛られた円卓の回りに5人の哲学者が座っており，それぞれの前に皿が1枚置かれ，その皿の両側にフォークが1本置かれる．各哲学者から見ると，左右に1本ずつフォークが置かれているように見えるが，テーブル全体では5本しかフォークはない．哲学者は2本のフォークを使って食事をしなければならないとして，全員が1本ずつフォークを取ってしまうと，誰も永久に食事ができなくなってしまう．

安全性の破壊誤りは，プロセス間で利用される共有資源の排他制御の失敗によって，データの一貫性が失われる誤りである．たとえば，2つのプロセスで同じ変数の値（仮に x=0）を読み込み，一方のプロセスが1加え（x=x+1），もう一方のプロセスが2加えて（x=x+2）から値を書き込むとする．この場合，値を同時に読み込んでしまうと，後に書き込んだ値だけが有効になるので，結果が一意に定まらない状況が生じる（x=1 or 2 or 3）．そのため，並列処理が行われているプログラムのテストでは，同時期アクセスが適切に制御されているかどうかを確認するために，さまざまなタイミングでのデータの読み書きのテストが要求される．

公平性の破壊誤りは，スタベーションや飢餓状態と呼ばれる，あるプロセスだけが実行されない誤りである．たとえば，「読み書き問題」がある．ある値を読み込むプロセス（Reader）が複数存在しており，その値に書き込むプロセス（Writer）が1つ存在する（複数でもよい）場合に，安全性の破壊誤りを起こさないために，Reader は，書き込みをしていないときだけ読み込むことができ，Writer は，誰も読み出していないときだけ書き込むことができるようにしなければならない．このとき，Reader が次々に読み込みを行うと，Writer はいつまでも書き込むことができなくなってしまう．

以降の章では，並列プログラム特有の同期誤りの中から特に，生存性の破壊誤りであるデッドロックの検出手法について，並列プログラムのモデル化と併せて述べる．また，安全性の破壊誤りと関連する競合状態のテスト手法について述べる．

並列プログラムのモデル化とデッドロックの検出

この章では，並列プログラムの動作のモデル化と生存性の破壊誤りであるデッドロックの検出法の一例として，並行状態グラフ[3]，および，事象相互作用グラフと協調路[4]について説明する．

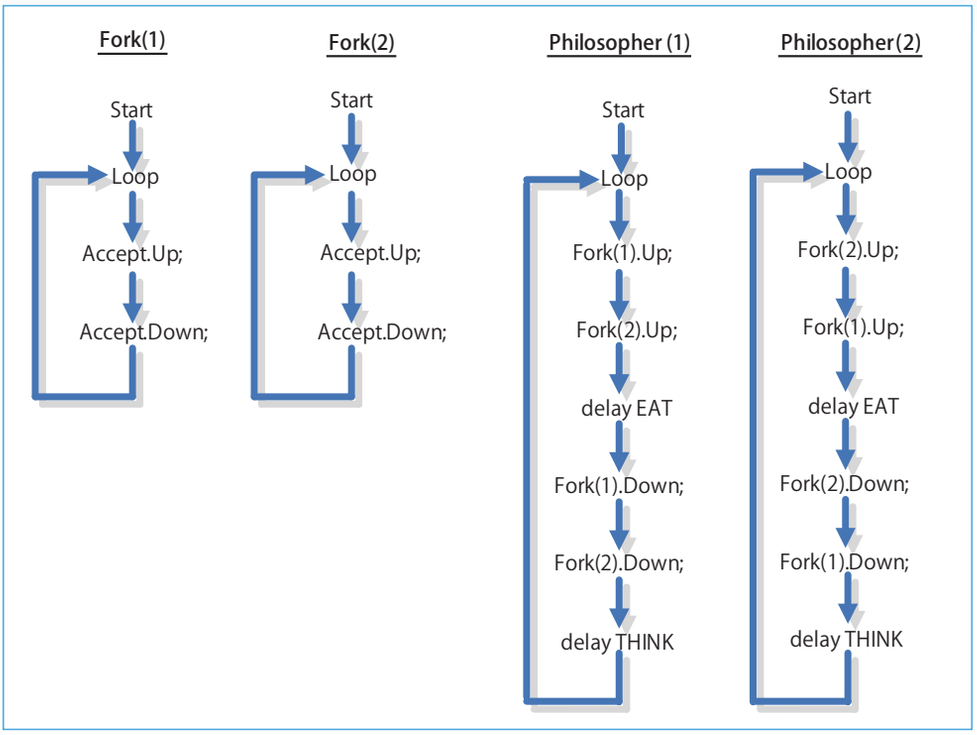

図 -1
2 人の哲学者食事問題のコントロールフローグラフ

例として，哲学者の食事問題を簡略化した 2 人の哲学者食事問題を用いる．**図 -1** に，2 人の哲学者食事問題のコントロールフローグラフを示す．哲学者を表すプログラム単位 Philosopher が 2 つあり，食事の際にはフォークを 2 本順に取り，食事が終わると使ったフォークを置いて瞑想する．フォークを表すプログラム単位 Fork が 2 つあり，上げ下ろしされる．Philosopher と Fork は，Up と Down の際に同期をとる．グラフ中の Up がフォークを持ったことを表し，Down がフォークを置いたことを表す．

[並行状態グラフ]

並列プログラムの動作を解析するためのモデル化として，並行状態グラフがある[3]．並列プログラムとは，逐次的に実行されるプロセスが互いに影響を及ぼし合うと捉えることによって，並列プログラム全体の状態を，各プロセスの状態の組合せによって定義したものである．並列プログラムを構成する個々のプロセスの状態の組合せを並行状態と定義する．並行状態は，プロセスの状態を変化させる操作によって遷移する．並行状態グラフは，並行状態を点とし，遷移を辺とするグラフで構成される．

図 -2 に，2 人の哲学者食事問題における並行状態グラフを示す．グラフにおいて，点は，各並行状態を表し，辺は制御の流れを表す．この例の場合，プログラムが (1, 2, 7) や (1, 2, 3, 4, 5, 13, 7) というパスを辿るとデッドロックに至るということが分かる．つまり状態 7 は，2 人の哲学者が各々フォークを 1 本ずつ持っている状態なので，次の状態に遷移できなくなる．

また，**表 -1** に，各並行状態の詳細を示す．この表は

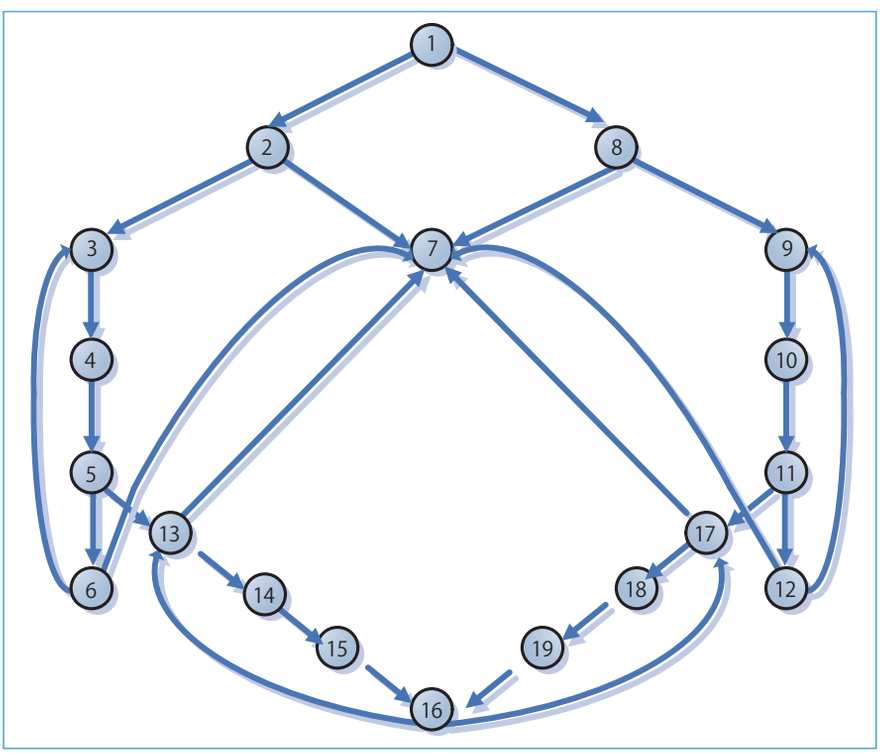

図 -2　2 人の哲学者食事問題の並行状態グラフ

5 並列プログラムのテスト

Concurrency State No.	Fork(1)	Fork(2)	Philos(1)	Philos(2)	Next States
1	Accept.Up	Accept.Up	Fork(1).Up	Fork(2).Up	2, 8
2	Accept.Down	Accept.Up	Fork(2).Up	Fork(2).Up	3, 7
3	Accept.Down	Accept.Down	Fork(1).Down	Fork(2).Up	4
4	Accept.Up	Accept.Down	Fork(2).Down	Fork(2).Up	5
5	Accept.Up	Accept.Up	Fork(1).Up	Fork(2).Up	6, 13
6	Accept.Down	Accept.Up	Fork(2).Up	Fork(2).Up	3, 7
7	Accept.Down	Accept.Down	Fork(2).Up	Fork(1).Up	No state
8	Accept.Up	Accept.Down	Fork(1).Up	Fork(1).Up	7, 9
9	Accept.Down	Accept.Down	Fork(1).Up	Fork(2).Down	10
10	Accept.Down	Accept.Up	Fork(1).Up	Fork(1).Down	11
11	Accept.Up	Accept.Up	Fork(1).Up	Fork(2).Up	12, 17
12	Accept.Up	Accept.Down	Fork(1).Up	Fork(1).Up	7, 9
13	Accept.Up	Accept.Down	Fork(1).Up	Fork(1).Up	7, 14
14	Accept.Down	Accept.Down	Fork(1).Up	Fork(2).Down	15
15	Accept.Down	Accept.Up	Fork(1).Up	Fork(1).Down	16
16	Accept.Up	Accept.Up	Fork(1).Up	Fork(2).Up	13, 17
17	Accept.Down	Accept.Up	Fork(2).Up	Fork(2).Up	7, 18
18	Accept.Down	Accept.Down	Fork(1).Down	Fork(2).Up	19
19	Accept.Up	Accept.Down	Fork(2).Down	Fork(2).Up	16

表 -1
並行状態グラフにおける並行状態の表

一般的な状態遷移表と同様の形式で書くことができ，並列プログラムの状態が，イベントにより次にどのような状態に遷移できるかを表形式で表現できる．たとえば，状態 2 から状態 3 に遷移する場合は，哲学者 1 がフォーク 1 を持っており，さらにフォーク 2 を持とうとしているので，哲学者 2 はフォーク 2 を持とうとしても持つことはできない．その間に，哲学者 1 はフォーク 1 を置く（状態 3）．

この並行状態グラフを用いて，並列プログラムをモデル化し，テストが十分に行われたかどうかの評価のために，並行状態グラフの点（並行状態）や辺，あるいは，それらの列（パス）を測定対象とするテスト基準を考えることができる．また，生存性の破壊であるデッドロックを論理的に検出でき，デッドロック状態に陥った場合にプログラムが適切に処理しているかをテストできる．

一方で，並行状態グラフには，並列プログラムの中のプロセス数が固定されていない場合，並行状態の数が組合せ論的に増大する，といった問題点がある．たとえば，図 -2 の場合，状態数は 19 で済むが，哲学者が 7 人になった場合（プロセス数が最大 7 になった場合）は，状態数が 32,063 になる．この場合，すべてのパスを網羅するテストケース数が，実際のテストにおいて実行不可能な数であることは想像に難くない．そのため適切なパスを設定し，テストケース数を削減することが必要となる．

逐次プログラムにおける制御パステストにおいて，命令文を少なくとも 1 回は実行するステートメントテストや，分岐を少なくとも 1 回は実行するブランチテストなど，ソフトウェアの品質目標に対してテスト基準を設けているように，並列プログラムにおいてもいくつか基準を設定し，品質目標に合わせ基準を適用することが可能である．

さらに，このような並列プログラムの場合は，基本的にプログラムが繰り返し同様な処理を行うことになる．そのため厳密なテストになると，無限回のループテストが必要となるため，なんらかの網羅基準が必要になる．

たとえば，有限なテストケースを生成する一番厳格な品質基準を適用するならば，有限な並行処理のパス（ループは除外する）を網羅するという意味で，all-proper-concurrent-histories という基準が設定できる．この基準は，プログラムのループ処理（(6, 3) に至るパス）を排除しているため，無限の網羅パステストから有限な網羅パステストにする．図 -2 では，(1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7) や，(1, 2, 7) などがパスになる．また，テストケース数を減少させるためにテスト基準を少し緩やかにし，グラフ上の辺だけを網羅するような all-edges-between-concurrent-state という基準も設定できる．この基準では，1 回実行された部分パス（edge）は実行しないという基準になるため，テストパスおよびテストケースの削減が可能になる．この基準では，(5, 6, 7) と (1, 2, 3, 4, 5) といった遷移が網羅される．結果的に，

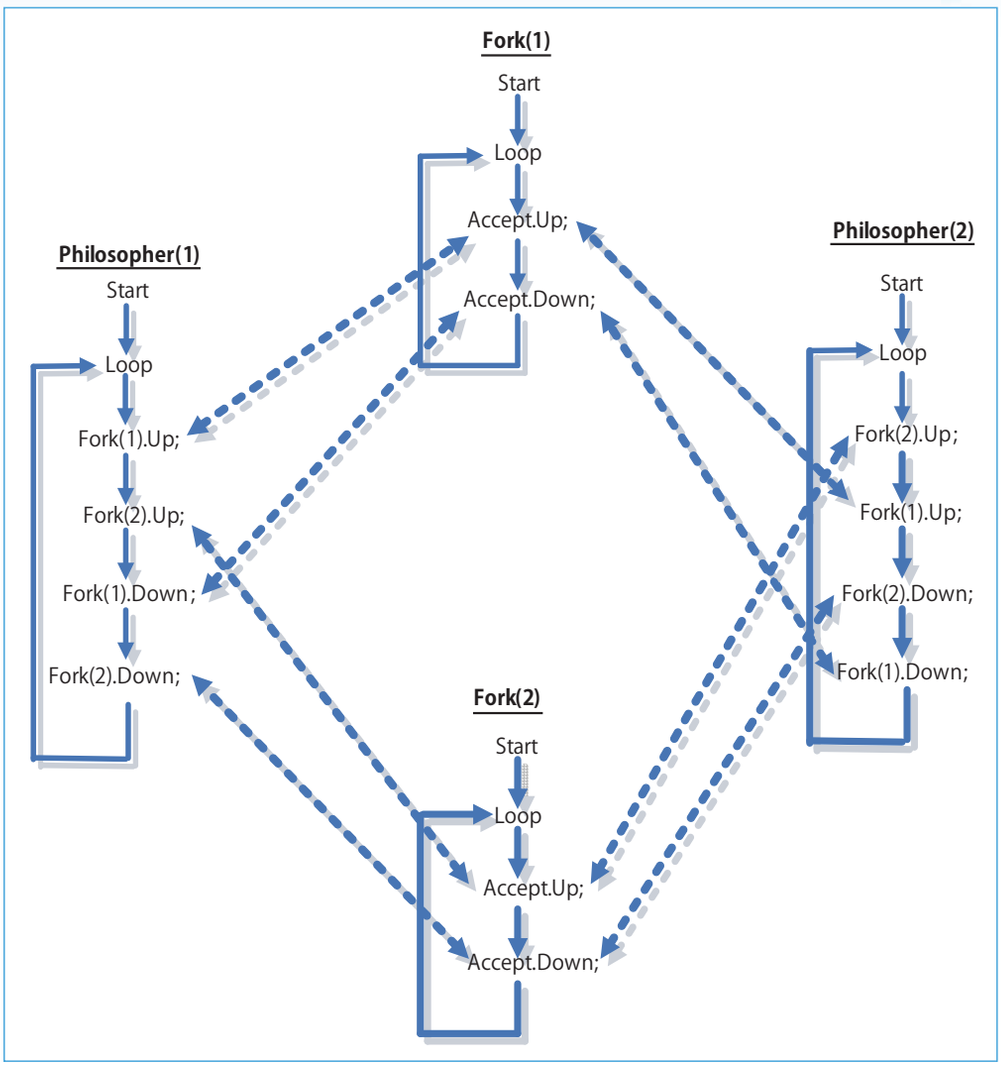

図 -3
2 人の哲学者食事問題の事象相互作用グラフ

(1, 2, 3, 4, 5, 6, 7) という長いテストパスを 2 つに分けることができることから，重複するパスの実行をする必要がなくなるので，テストケース数を削減できる．

［事象相互作用グラフと協調路］

並列プログラムを逐次的に動作するプロセスが並列に動作すると考え，その動作を事象相互作用グラフでモデル化する [4]．事象相互作用グラフは，事象グラフの集合と相互作用の集合である．プログラム単位ごとに，制御の流れを表すコントロールフローグラフを作成し，プロセス間で相互作用を行う文を，並行事象文と呼ぶ．並行事象文には，プロセスの生成や，同期や通信を行う文，共有資源を操作する文などが相当する．コントロールフローグラフ中の，並行事象文と並行事象文を含む判定文とで構成しなおしたグラフを，事象グラフと呼ぶ．相互作用は，2 つの事象グラフの中の並行事象文と相互作用名とで表す．

図 -3 に，2 人の哲学者食事問題における事象相互作用グラフを示す．この事象相互作用グラフは，4 つの事象グラフからなり，8 つの相互作用（同期）が存在する．事象相互作用グラフにおいて，点は，並行事象文と並行事象文を含む判定文で表す．図 -3 では，同期をとる Up と Down を含む文が並行事象文であり，この並行事象文を含む loop 文が並行事象文を含む判定文として，事象グラフは構成される．また，実線の辺は，コントロールフローグラフと同じ制御の流れを表しており，破線の辺が相互作用（同期）を表している．この例では，Philosopher と Fork 間で Up 同士と Down 同士が破線の辺で結ばれ，この間で同期をとることを表している．

この事象相互作用グラフ上で協調路を作成する．協調路とは，事象グラフそれぞれから実行パスを取り出し，任意の 2 つの実行パスの間で相互作用の個数が同じであるよう組み合わせたものである．すなわち，並列プログラムが m 個のプログラム単位からなる場合，協調路は，m 個の実行パスの組となる．

図 -4 は，2 人の哲学者食事問題における協調路の一例である．この例では，プログラム単位が 4 個であるので，4 つの事象グラフそれぞれから取り出した，4 個の実行パスの組で協調路が構成される．また，2 つの実行パス間で相互作用の出現個数が同じであり，それぞれ

5 並列プログラムのテスト

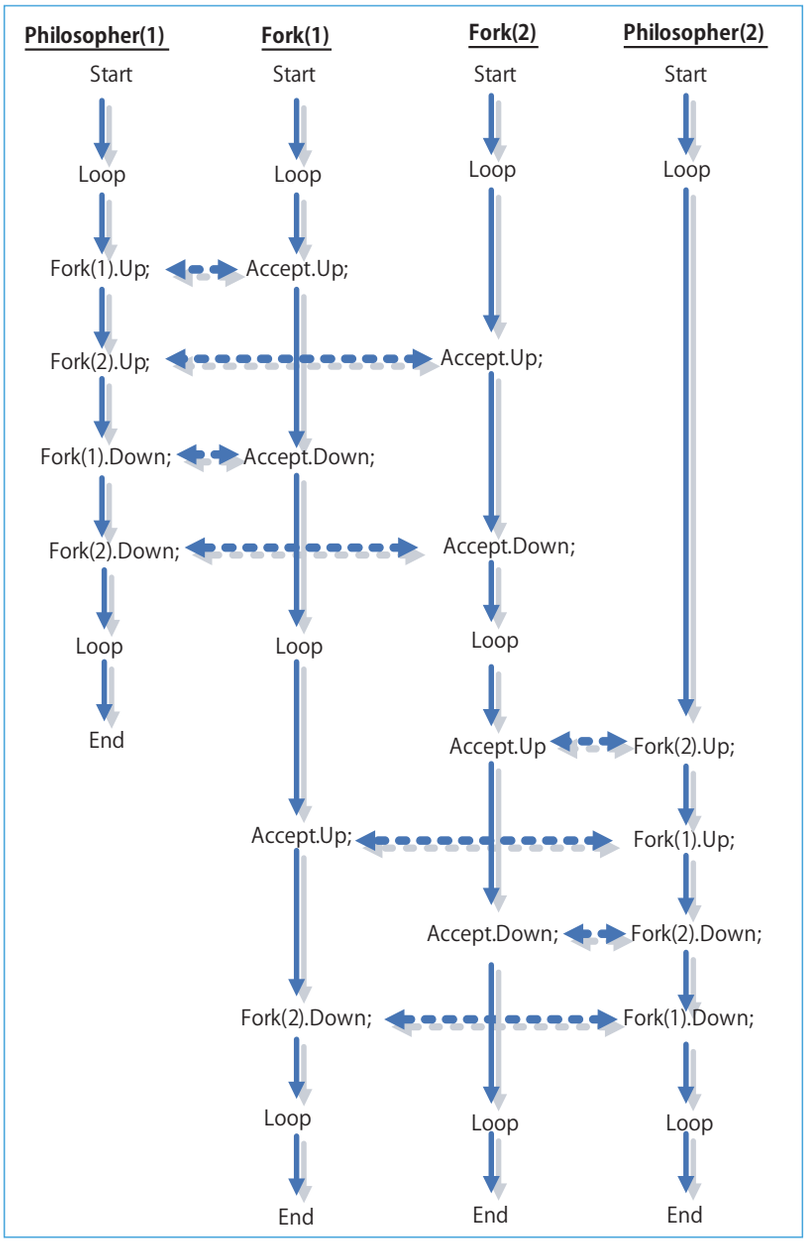

図 -4　2 人の哲学者食事問題における協調路の一例

の時点で同期をとりながら，各プロセスが処理を進めていくことを表している．

事象相互作用グラフと協調路とを利用することによって，逐次プログラムにおけるテスト基準である，ステートメントテストや，ブランチテストに加えて，並行事象文を少なくとも 1 回は実行するなどの基準を設けて，テストを実施することができる．また，作成した協調路において，相互作用の出現順序が実行パスで矛盾する場合，デッドロックの発生を検出できる．

図 -5 は，2 人の哲学者食事問題におけるデッドロックを示す協調路の一例である．図において，相互作用（同期）を表す破線の辺の交差が 2 カ所で起きており，同期をとるために待ち状態が発生する．このため，各プロセスが処理を進めることができなくなり，デッドロックとなることが分かる．

競合状態のテスト手法

この章では，安全性の破壊誤りと関連する競合状態のテスト手法について説明する．以下の条件を共に満たしている場合には，競合状態が発生する可能性があり，安全性の破壊誤りが起こり得る．

- 少なくとも 1 つの変数が書き込み可である
- 複数のスレッドがその変数に対し同時期のアクセスを禁止する仕組みを提供していない

このようなバグは，データのロックや，データの更新タイミングを考慮しないために起こる場合が多い．たとえば**図 -6** に示すように，2 つのスレッドが十分時間をおいてデータにアクセスしている場合は問題が起こらないが，2 つのスレッドがほぼ同時期にデータにアクセスすることにより，そのデータがロックされてしまったり，正しく更新されないことが起こる．

このため，プログラムを記述する際には，Lamport による happens-before の関係（たとえばスレッド 1 は必ずスレッド 2 の後に呼ばれる，など）を定義したりし，共有変数などが適切なタイミングでアクセスされることを保証しなければならない．これらをテストするためには，静的にプログラム構造を分析する方法と，動的にプログラムを実行させテストする方法がある．

静的な場合は，プログラムの構造を解析することになる．多くの研究やツールはプログラム中から競合状態に陥る可能性の高い変数を抽出し，その変数が安全性の破壊誤りを起こすかどうかを確認する．しかし静的解析の場合は非常に多様な競合状態があるため，ただ単に共有変数を抽出するだけでは，正確に解析できない．

たとえば 2 つのプロセスがあり，かつ，その 2 つのプロセスが共有変数を持っていたとしても，その共有変数に対して 2 つのプロセスが読み込みのみのアクセスしか起こさない場合には，安全性の破壊誤りは起きない．また，2 つのプロセスがあり，かつ，その 2 つのプロセスが同時期に実行されないのであれば，たとえ共有変数に値が書き込まれても，変数は競合状態が起きない．これらさまざまな例外条件をフィルタリングすることによって，静的に安全性の破壊誤りを検出することが可能になる [5]．

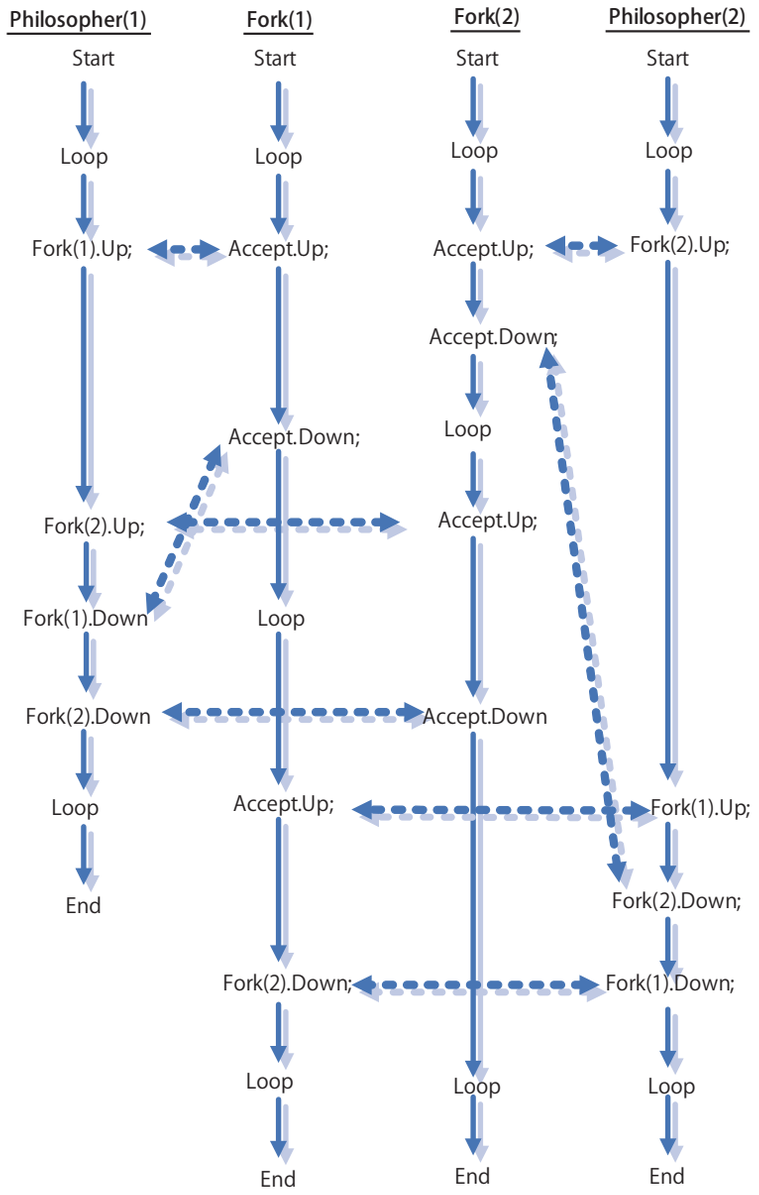

図 -5　2 人の哲学者食事問題におけるデッドロックを示す協調路の一例

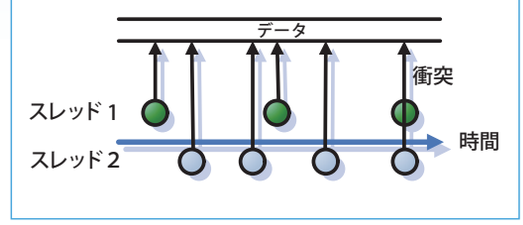

図 -6　マルチスレッドの衝突

動的テストを実行する場合は，さまざまなタイミングで並列プログラムを実行させることによって，再現しにくい並列プログラム特有のタイミング依存のバグを検出する．そのような場合は，Java マルチスレッドのテストフレームワークとして提案されている ConTest[6] などを使いテストすることが可能である．以下，ConTest について述べる．

ConTest は，Java のマルチスレッドの動的テストやデバッグ，カバレッジの測定に使用するツールである [6]．ConTest の基本的な原理は，計測段階でクラスファイルを変換し，ConTest のランタイム機能への呼び出しを，特定の選択された場所に挿入する．ConTest を実行すると，この特定の場所で，コンテキスト・スイッチ（実行プロセスの切り替えなど）を発生させる．この特定の場所とは，同期ブロックの出入口や共有変数へのアクセスなど，スレッド間の相対順序が実行結果に影響を与える可能性のある場所である．すなわち，yield() や sleep() などのメソッドを呼び出すことにより，コンテキスト・スイッチが実行される．この処理をランダムに実行させるため，実行のたびに異なったタイミングでの実施が可能となり，さまざまなタイミングで起こる安全性の破壊誤りのバグが再現可能となる．

ConTest は，他にもデッドロック防止機能を持つ．ConTest は，矛盾した順序でロックがネストされていないかどうかを分析することができ，こうした状態を発見することにより，デッドロックを引き起こす危険性を検知できる．ただしこの分析は，テストを実行した後に，オフラインで実行する．

ConTest を利用することによって，安全性の破壊誤りに着目したテストを動的に実施でき，さまざまなタイミングで起こる安全性の破壊誤りのバグを再現可能になる．また生存性の破壊誤りであるデッドロックの防止も期待できる．

まとめと今後の展望

本稿では，並列プログラムのテストの難しさの原因とテスト手法について述べた．並列プログラムの特性と誤りの分類について述べ，その中から特に，デッドロックの検出手法と競合状態のテスト手法について述べた．

並列処理が持つ本質的な難しさやその原因については，1980 年代からすでに指摘されており，現在もそれほど変わっていないと言える．一方，ここにきて，マルチコアやマルチスレッドに代表されるように，並列プログラムが望まれる背景が整ってきた．今後，多くのシステムやソフトウェアが，マルチコアに対応すると予想される．

しかし，現在多くの CPU がマルチコア化されているにもかかわらず，ソフトウェアの処理スピードがそれに

5 並列プログラムのテスト

対応した伸びをみせていない．なぜなら，開発者がマルチコアの特性というべきプログラムの並列処理に消極的なためである．多くの開発者は，ドライバーソフトウェアなどスレッドに分けやすい部分の処理だけを並列処理化し，他の部分の処理については，シングルコアと同様な処理をしている．このような状況は，開発者自身の並列プログラムについてのスキル不足に問題が起因するが，並列処理のテストとデバックの困難さもその遠因である．

今後 CPU 性能を最大限活かすために，プログラムのより多くの部分で並列処理をする必要が生じることは自明である．そのため，並列プログラムに特化した動作のモデル化やテスト設計手法，テスト実行環境など，並列プログラムのテスト手法について研究を進めることは重要である．また，並列処理開発環境を支援するためのデバッガや ConTest のようなテストツールが必要となると考える．

参考文献

1）古川善吾，伊東栄典，片山徹郎：並行処理プログラムの試験，情報処理，Vol.39, No.1, pp.7-12 (Jan. 1998).
2）Ben-Ari, M. : Principles of Concurrent Programming, Prentice Hall (1982).
3）Taylor, R. N., Levine, D. L. and Kelly, C. D. : Structural Testing of Concurrent Programs, IEEE Trans. Softw. Eng., Vol.18, No.3, pp.206-215 (1992).
4）片山徹郎，菰田敏行，古川善吾，牛島和夫：並行処理プログラムにおけるテストケースの定義と生成ツールの試作，情報処理学会論文誌，Vol.34, No.11, pp.2223-2232 (1993).
5）Naik, M., Aiken, A. and Whaley, J. : Effective Static Race Detection for Java, Proc. 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp.308-319 (2006).
6）Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G. and Ur, S. : Framework for Testing Multi-threaded Java Programs, Concurrency and Computation : Practice and Experience, John Wiley & Sons, Vol.15, Issue 3-5, pp.485-499 (2003).

（平成 19 年 12 月 29 日受付）

片山徹郎（正会員）

kat@cs.miyazaki-u.ac.jp

1996 年九州大学大学院工学研究科情報工学専攻博士後期課程修了．博士（工学）．同年奈良先端科学技術大学院大学情報科学研究科助手に採用．2000 年宮崎大学工学部情報システム工学科助教授に採用．並列プログラムや組込みシステムを対象としたテスト手法についての研究に従事．

高橋寿一

juichi@ieee.org

2000 年フロリダ工科大学コンピュータサイエンス学部ソフトウェアエンジニアリング学科修士課程修了．2003 年広島市立大学大学院情報科学研究科情報科学専攻博士後期課程修了．ソニー（株）現職．ディスティングイッシュ・エンジニア．ソフトウェアテスト，分散処理システムの研究開発に従事．博士（情報工学）．IEEE-CS，ACM，日本品質管理学会各会員．