分散計算における制御フローに基づいたイベントアブストラクション手法

多田知正[†] 樋口昌宏[††]

本論文では分散計算の解析を容易にすることを目的とし，各プロセスにおける制御フローに着目して分散計算の構造を簡略化する手法について述べる．分散計算は，一般にそれぞれのプロセスで発生するイベントの半順序集合として表すことができる．大規模な分散計算の解析を容易にすることを目的として，複数のイベントを1つの抽象イベントに置き換えることによって計算全体の構造を簡単化するイベントアブストラクションという手法が提案されている．従来提案されている手法では，解析者が計算全体の構造を把握する必要があるため，解析者の負荷が大きいという問題がある．提案手法では，イベント間の因果関係を制御依存関係とデータ依存関係に分類し，制御依存関係に基づき計算中の閉じた制御フローを抽出する．1つの閉じた制御フローは分散計算中のひとまとまりの処理と考え，これを1つの抽象イベントに置き換えることにより計算全体の構造を簡略化する．提案手法の有効性を議論するため，実際にイベントアブストラクションを行い，結果を可視化するツールを作成した．そして，いくつかの分散プログラムの実行結果に作成したツールを適用し，バグの検出に一定の有効性があることを確認した．

An Event Abstraction Method in Distributed Computations Based on Control Flow

Harumasa Tada[†] and Masahiro Higuchi[††]

In this paper, we propose a simplification method of structure of distributed computations. In general, distributed computations are modeled as partial ordered events. In order to facilitate analysis of distributed computations, event abstraction has been proposed. Event abstaction reduces apparent complexity of a distributed computation by replacing some related events with one abstract event. The existing method of event abstraction is difficult to use because an analyzer should grasp the structure of the distributed computation. We classified the causal relation of events into two types, that is, the control dependency and the data dependency. Then, we defined the control flow based on the control dependency. In the proposed method, closed control flows are extracted from the distributed computation. Since each closed control flow can be considered as a module in the execution, events involved in a closed control flow are replaced with an abstracted event. To evaluate the proposed method, we implemented a visualization tool of distributed computations based on the method. We applied the tool to some of distributed programs and showed that the proposed method is useful for detecting some kind of programming errors.

1. まえがき

分散システムとは，複数のプロセスがメモリを共有せず，ネットワークを用いて通信しながら情報を交換し，並列に処理を行うシステムである．各プロセスは逐次的な実行を行っており，実行の単位をイベントという．分散システムの計算（分散計算）は，各プロセスで発生したイベントの集合と，イベント間の因果関係によりモデル化できる[8]．このようなイベントベースの分散計算モデルは，分散システムの設計，解析，デバッグなどに広く用いられている[1),3)~5),7),9),11)]．

分散計算は一般に非決定的であるため，分散計算を解析する際には，実行時に生成されたログを解析する方法が通常用いられる．しかし，分散計算のログは，その中に含まれるイベント量の膨大さや，イベント間の因果構造の複雑さといった要因から，解析が非常に困難である．

そのため，複数のイベントを1つの抽象イベントに置き換え，その内部の因果関係を隠蔽することで，分散計算の構造を簡単化する，イベントアブストラクションという手法が提案されている[6),7)]．従来のイベ

† 大阪大学大学院基礎工学研究科
Graduate School of Engineering Scicence, Osaka University
†† 近畿大学理工学部
School of Science and Engineering, Kinki University

ントアブストラクション手法は，あらかじめ解析者が記述したテンプレートに一致するイベント群を抽出し，これをアブストラクションするものである．しかし，複数のプロセスに及ぶ計算の構造を把握しながらテンプレートを記述することは一般に非常に困難である．

そこで本研究では，イベント間の因果関係を，制御依存関係とデータ依存関係に分類し，制御依存関係に基づいたイベントアブストラクション手法を提案する．イベント間の制御依存関係とは，イベントの前後関係のみで定義される因果関係とは異なり，制御の流れを考慮して定義される．本手法は，制御依存関係に基づいて定義される閉じた制御フローを分散計算より抽出し，その中に含まれるイベントを 1 つの抽象イベントに置き換えるものである．この手法では，制御依存関係の情報をログに記録しておくだけでイベントアブストラクションが可能であるため，従来の手法と比べて解析者の負荷は大幅に軽減される．

本研究では，実際に提案手法を適用した分散計算の可視化ツールを作成した．また，分散プログラムを入力として，分散計算のログを生成するようなシミュレータを作成し，シミュレータの生成したログを可視化ツールに入力して実験を行い，提案手法の評価を与えた．

以下，2 章では，本研究で扱う分散計算のモデルについて述べる．3 章では，イベントアブストラクションの概略と，従来の手法について述べる．4 章で，提案手法の説明を行い，5 章では，作成した可視化ツールを用いて提案手法の評価を行っている．最後に 6 章で結論を述べる．

2. 準　　備

分散システムは非同期通信を行う複数のプロセスから構成されている．それぞれのプロセスは逐次的に実行を行っており，プロセスの実行単位をイベントと呼ぶ．イベントは

(1) 他のプロセスにメッセージを送信する送信イベント
(2) 他のプロセスが送信したメッセージを受信する受信イベント
(3) (1), (2) 以外のプロセス内部で実行される内部イベント

に分類される．

あるプロセスの実行結果を表すイベントの系列をそのプロセスの計算という．分散システムを構成する個々のプロセスの計算の集合を分散計算という．分散計算において，2 つのイベント a，b の間に以下のいずれかが成り立つとき，a，b 間には因果関係（causality relation）が存在するといい，$a \prec b$ と書く．

(1) a と b は同一プロセスで発生しており，かつ a が b より先に発生している．
(2) a がメッセージ m の送信イベントであり，b が m の受信イベントである．
(3) $a \prec c$ かつ $c \prec b$ なるイベント c が存在する．

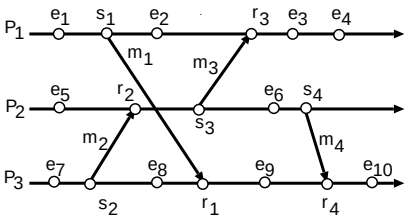

図 1 分散計算の例
Fig. 1 An example of distributed computation.

分散計算は，イベントの集合 E と，因果関係 $\prec$ でモデル化できる．具体例を**図 1** に示す．P_1〜P_3 はプロセスであり，横方向の矢線は時間の経過を表している．頂点はイベントに対応しており，e_1〜e_{10} は内部イベント，s_1〜s_5 は送信イベント，r_1〜r_5 は受信イベントである．m_1〜m_5 はメッセージである．このようにイベントとその因果関係をもとにした分散計算のモデルをイベントベースの分散計算モデルと呼ぶ[2]．

3. イベントアブストラクション手法

3.1 イベントアブストラクション

分散計算の各プロセスの実行したイベントを記録したファイルをログと呼ぶ．通常ログに記録された個々のイベントにはベクトルタイムスタンプ[10]というイベントの論理的な時間を示すタイムスタンプが付加されており，これによりイベント間の因果関係を求めることができる．

分散計算はその非決定的な性質から，分散システムの実行後にログを用いて解析を行う方法が一般的である．しかし，イベントベースのモデルで表現された分散計算は，その中に含まれるイベント量の膨大さや，それに起因するイベント間の因果関係の複雑さといった要因から，解析が非常に困難である．

そこで，分散計算の複数のイベントをまとめて 1 つの抽象イベントとし，内部の因果関係を隠蔽することで計算全体の構造を単純化するような，イベントアブストラクションという手法が提案されている．

3.2 従来の手法と問題点

従来のイベントアブストラクション手法は，ログにそれぞれのイベントに関する詳細な情報を付加し，そ

の付加情報をもとにイベントアブストラクションを行う．

Bates [7]の手法では，イベントはあらかじめいくつかのクラス（例：e_openfile, e_readfile など）に分類されている．解析者はアブストラクションしたいイベント群のテンプレートを記述する．テンプレートは，イベントのクラス名をアルファベットとし，逐次実行，選択，並行実行，繰返しを表す演算子を用いて記述されるイベント式と，それに付随する制約条件（例："イベント式中の e_openfile の属性 fileID が e_readfile の fileID と等しい" など）からなる．アブストラクションの際には，計算全体のログの中で，テンプレートにマッチする部分を抜き出し，これを 1 つの抽象イベントに置き換える．この方法は，ログの中から目的とする部分を抽出するためには有効であるが，アブストラクションしたいイベント群のパターンをあらかじめテンプレートとして記述しなければならず，分散計算全体の構造を把握したいような場合には解析者の負荷が大きく，不向きである．また，複数のプロセスに点在するイベント群をアブストラクションするためには，テンプレートを記述する際，複数のプロセスにわたる計算の構造を把握する必要がある．この構造は一般に半順序であり，解析者がこれを正しく把握し，記述することは非常に困難である．

4. 制御フローに基づくイベントアブストラクション手法

本研究では，制御の流れに着目してイベントアブストラクションを行うことを考えた．ある単一プロセスにおいてはイベントは逐次的に実行されているので，制御の流れは基本的に線形であり，制御の流れが分岐，合流するのは，各プロセスがメッセージにより情報交換を行うためである．本研究ではメッセージの送受信時に着目し，制御の流れに基づくイベント間の関係を定義する．

プロセス P がメッセージ m_1 を送信する際，m_1 の送信イベント s_1 は，P において s_1 に続くイベント e_1 と，m_1 の受信イベント r_1 の両方に影響を及ぼす可能性がある．このため，制御の流れはここで分岐する場合がある．また，プロセス P がメッセージ m_2 を受信する際，その受信イベント r_2 は m_2 の送信イベント s_2 と，直前のイベント e_2 から影響を受ける可能性があるので，制御の流れはここで合流する場合がある．

ここで，**図 2** のような場合を考える．プロセス P はあるメッセージ m を受信した．そこで，今までの処理をイベント a で一時中断し，受信したメッセージの処理を行い，その処理が終了した後で中断していた処理を再開させた（イベント b）とする．このような場合を割込み受信という．このとき，因果関係では $a \prec r$ であるが，実際には a は r に直接影響を与えておらず，制御の流れは a から r ではなく b に続いている．このように，イベント間の因果関係は，必ずしも各プロセスにおける制御の流れを反映しているとはいえない．そこで本研究では，因果関係のうち，制御の流れを反映しているものを，制御依存関係（$\overset{c}{\rightsquigarrow}$）とし，そうでないものを，データ依存関係（$\overset{d}{\rightsquigarrow}$）として区別する．つまり，上記のような場合，$a \overset{c}{\rightsquigarrow} b$, $a \overset{d}{\rightsquigarrow} r$ となる．

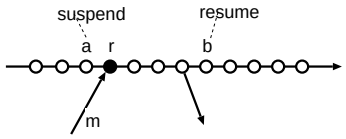

図 2 割込み受信
Fig. 2 Interruption by receiving.

制御依存関係を定義するには，イベントを従来より詳細に分類する必要がある．通常の内部イベントのほかに，以下の 3 種類の内部イベントを追加する．

中断イベント プロセスが以前の処理を何らかの理由で中断し，別の処理を行う際に実行される．

復帰イベント プロセスが中断していた処理を再開する際に実行される．

終了イベント プロセスが現在の処理を終了する際に実行される．

さらに送信イベントと受信イベントをそれぞれ 2 種類に分類する．

分岐送信イベント プロセスがメッセージを送信したのち，現在の処理を引続き行う場合に実行され，制御の流れはここで 2 つに分岐する．

非分岐送信イベント プロセスがメッセージを送信することで，制御を別のプロセスに移し，自身は別の処理を行う場合に実行される．

合流受信イベント プロセスが受信したメッセージがそれまでの処理に関するものである場合に実行され，制御の流れはここで合流する．

非合流受信イベント プロセスが受信したメッセージがそれまでの処理に関するものでない場合に実行される．

非合流受信イベントおよび復帰イベントの直前のイベントは，中断イベント，終了イベント，非分岐送信イベントのいずれかである．

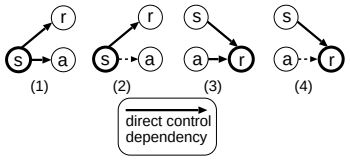

図 3 送受信イベント付近の直接制御依存関係
Fig. 3 Direct control dependency of send/receive events.

4.1 制御依存関係とデータ依存関係

ここで，イベント間の制御依存関係（control dependency）とデータ依存関係（data dependency）を定義する．まず，イベント間の直接制御依存関係（direct control dependency）を定義する．

定義 1 直接制御依存関係（$\xrightarrow{c}$）は以下により定義される最小の関係である．

- 送信イベント s と s に対応する受信イベント r の間に $s \xrightarrow{c} r$ が成り立つ．
- 中断イベント x と x に対応する復帰イベント y の間に $x \xrightarrow{c} y$ が成り立つ．
- 分岐送信イベント s とその直後のイベント e の間に $s \xrightarrow{c} e$ が成り立つ．
- 受信イベント r とその直後のイベント e の間に $r \xrightarrow{c} e$ が成り立つ．
- 内部イベント（中断イベントと終了イベントを除く）i と，その直後のイベント e の間に $i \xrightarrow{c} e$ が成り立つ．

また，制御依存関係（$\overset{c}{\rightsquigarrow}$）は，直接制御依存関係の反射推移閉包として定義される． □

送受信イベント付近の直接制御依存関係を，**図 3** に示す．図中で，s は送信イベント，r は受信イベントを表す．

- (1) の s は分岐送信イベントである．このとき，$s \xrightarrow{c} e$，$s \xrightarrow{c} r$ が成り立つ．
- (2) の s は非分岐送信イベントである．このとき，$s \xrightarrow{c} r$ のみが成り立つ．
- (3) の r は合流受信イベントである．このとき，$e \xrightarrow{c} r$，$s \xrightarrow{c} r$ が成り立つ．
- (4) の r は非合流受信イベントである．このとき，e は中断イベント，終了イベント，非分岐送信イベントのいずれかである．よって，$s \xrightarrow{c} r$ のみが成り立つ．

イベント間の直接制御依存関係は，そのイベントの発生したプロセスにおいて局所的に決定できる．したがって，直接制御依存関係の情報をログに記録することは容易であり，これによりログからイベントの制御依存関係を構成することができる．

定義 2 2 つのイベント a，b について，$a \prec_i b$ であり，かつ $a \overset{c}{\rightsquigarrow} b$ でないとき，a，b 間にはデータ依存関係が成り立ち，$a \overset{d}{\rightsquigarrow} b$ と書く． □

$a \overset{d}{\rightsquigarrow} b$ である場合，a，b 間には直接的な制御の流れはないものの，プロセスの内部状態や，プロセスの管理するデータを介して，a が b に間接的に影響を及ぼしている可能性があることを意味している．

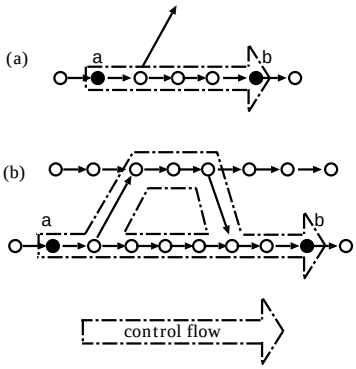

図 4 制御フローの例
Fig. 4 Examples of control flow.

4.2 閉じた制御フロー

$a \overset{c}{\rightsquigarrow} b$ なる 2 つのイベント a, b 間には，直接制御依存関係とイベントによる経路が存在する．これを a から b への制御フローと呼ぶ．このときイベント a を先頭イベント，b を終端イベントと呼ぶ．**図 4** に制御フローの例を示す．図 4 (b) で示されるように a から b への制御フローは，途中で分岐，合流する場合がある．制御フローは形式的にはイベントの集合として定義される．

定義 3 イベント a から b への制御フロー F は次のように定義されるイベントの集合である．

$$F = \{x \mid a \overset{c}{\rightsquigarrow} x \wedge x \overset{c}{\rightsquigarrow} b\}$$

□

定義 4 イベント a から b への制御フロー F が次の条件を満たすとき，F を閉じた制御フローという．

$$\forall x \forall y((x \xrightarrow{c} y) \Rightarrow (x \in F \Leftrightarrow y \in F))$$

□

4.3 制御依存関係に基づくイベントアブストラクション

ある 2 つのイベント間の制御フローが閉じているとき，制御フローに含まれているイベントは，制御フローの外のイベントとの間には制御依存関係を持たない．つまり，制御フローが閉じている場合，その制御フローは一連のまとまった処理であると見なすことができる．よって，閉じた制御フローを検出し，その中に含まれるイベントを 1 つの抽象イベントに置き換え

ることでイベントアブストラクションを行う．

閉じた制御フローは，計算のログに直接制御依存関係に関する情報が付加されていれば，機械的に検出することが可能である．また 4.1 節で述べたように，直接制御依存関係はイベントの発生したプロセスにおいて局所的に決定できる．このため，テンプレートを記述する際，複数のプロセスにわたる計算の構造を把握する必要のある従来の手法と比べて，解析者の負荷は大幅に軽減されると考えられる．

閉じた制御フローを検出し，イベントアブストラクションを行う単純な方法は以下のとおりである．最初に，閉じた制御フローの先頭イベントの候補 $HEAD = \{a \mid \neg\exists x(x \xrightarrow{c} a)\}$，および終端イベントの候補 $TAIL = \{b \mid \neg\exists x(b \xrightarrow{c} x)\}$ を求める．次に $HEAD$ の各要素 a から直接制御依存関係を深さ優先探索でたどり，見つかったイベントの集合を $CF(a)$ とする．また $TAIL$ の各要素 b から直接制御依存関係を逆向きに深さ優先探索でたどり，見つかったイベントの集合を $CP(b)$ とする．その後，$\exists b(CF(a) = CP(b))$ となるすべての a について $CF(a)$ に含まれるイベントを 1 つの抽象イベントに置き換える．定義 3，定義 4 より，これによって得られる抽象イベントは閉じた制御フローに対応している．しかしこの方法では，直接制御依存関係を順方向と逆方向の 2 通りの方法でたどる必要があり，また同じイベントを何度も探索する可能性がある．そこで各イベントを 1 度だけ探索することによりイベントアブストラクションを行うアルゴリズムを**図 5** に示す．図 5 において，イベント e が実行されたプロセスを $P(e)$ で表す．また全イベントの集合を E とする．

```
EventAbstraction
 1  HEAD ← {a | ¬∃x(x -c-> a)}
 2  foreach i ∈ E do
 3    R[i] ← null
 4  foreach a ∈ HEAD do
 5    JS ← φ
 6    JO ← φ
 7    TAIL ← φ
 8    F ← φ
 9    VS = TRUE
10    visit(a)
11    if VS ∧ (|TAIL| = 1) ∧ (JS = JO) then
12      F を 1 つの抽象イベントに置き換える
visit(e)
13  R[e] ← a
14  F ← F ∪ {e}
15  if ¬∃x(e -c-> x) then
16    TAIL ← TAIL ∪ {e}
17  else
18    foreach i such that e -c-> i do
19      if i が合流受信イベント then
20        if P(e) = P(i) then
21          JS ← JS ∪ {i}
22        else
23          JO ← JO ∪ {i}
24      if R[i] = null then
25        visit(i)
26      else if R[i] ≠ a then
27        VS = FALSE
```

図 5 イベントアブストラクションのアルゴリズム
Fig. 5 The algorithm for event abstraction.

4.4 分散プログラムへの適用

実際の分散計算において，制御フローがどのように構成されるかを示す．分散プログラムの例として，分散データベースのトランザクション処理において用いられる 2 つのプロトコルを実装したものを考える．

4.4.1 2 相コミット

2 相コミット（2PC）は，複数のプロセスで実行されるトランザクション T をコミット（正常終了）する際に，T を実行したすべてのプロセスの同意をとるためのプロトコルである．

2 相コミットの実行の様子を**図 6** の (a) に示す．T をコミットする際，T を開始したプロセス P_2 は，T を実行したすべてのプロセスに，T をコミットしてよいか問い合わせる．問い合わせたすべてのプロセスからコミット可能の返答が得られたときのみ P_2 は T をコミットでき，同意を求めた全プロセスにコミットすることを伝える．それを受けて，各プロセスはそれぞれ T のコミット処理を行い，処理が終了した後，P_2 にそのことを通知する．同意を求めたすべてのプロセスからコミット処理終了の返答が帰ってきた時点で T のコミット処理が完了する．

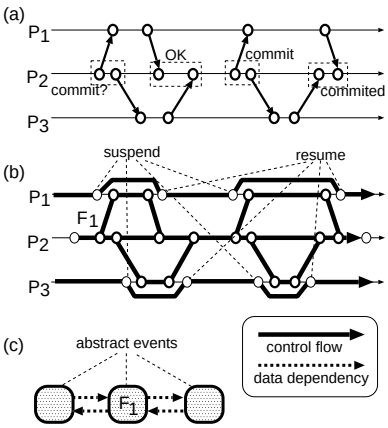

図 6 2 相コミット
Fig. 6 2 phase commit.

この場合の制御フローを，図 6 の (b) に示す．2 相コミットの開始から終了までの制御フロー F_1 が閉じていることが分かる．このように，2 相コミットの場

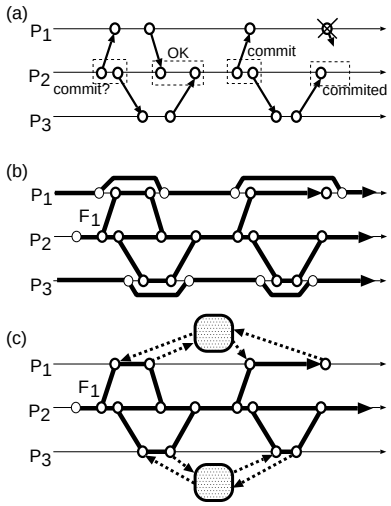

図 7 2 相コミット（バグのある場合）
Fig. 7 2 phase commit (with an error).

合通常閉じた制御フローとなる．P_2 における送信はすべて分岐送信イベント，受信はすべて合流受信イベントである．プロセス P_1，P_3 における制御フローは，P_2 からのメッセージを受信するたびに中断し，P_2 に返答を返した後に復帰している．イベントアブストラクションの結果は図 6 の (c) のようになる．

2 相コミットにおいて，バグのためにあるプロセス P_1 がコミットが終了したことを通知しなかったとする（**図 7** の (a)）．この場合 P_2 は P_1 からの返答を待ち続けることになり，コミット処理は終了しないため，制御フロー F_1 は閉じないことになる（図 7 の (b)）．したがって，イベントアブストラクションを行っても，この部分は抽象イベントに置き換えることができない（図 7 の (c)）．すなわちこの場合は，アブストラクションされなかった部分を分散プログラムのバグとして容易に検出できる．

4.4.2 2 相ロック

2 相ロックは，データベースにおいて，データの一貫性が破壊されるのを防ぐために，トランザクションがアクセスするデータをあらかじめロックして，他のトランザクションからのアクセスを禁止するプロトコルである．2 相ロックの実行例を**図 8** の (a) に示す．あるプロセス P_1 が実行中のトランザクション T が，あるデータ x を管理しているプロセス P_2 に x のロックを要求する．P_2 は x が他のトランザクションにロックされていないとき，x を T によってロックし，P_1 にそのことを伝える．ロックを獲得した P_1 は T の処理が終われば x のロックを解放する．P_2 はロック解放要求を P_1 から受け取り，x のロックを解放し，そ

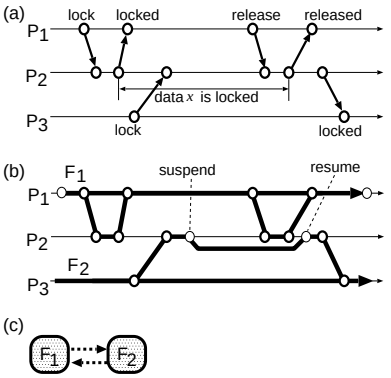

図 8 2 相ロック
Fig. 8 2 phase locking.

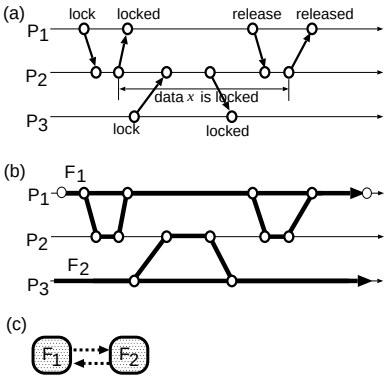

図 9 2 相ロック（バグのある場合）
Fig. 9 2 phase locking (with an error).

のことを P_1 に通知する．別のプロセス P_3 の出した x のロック要求は P_2 がロック要求を受け取った時点で x がすでにロックされていたので，ロックが解放されるまで待たされた後に受理されている．

この例における制御フローは図 8 の (b) のようになる．P_1 がロック要求を出してロックを獲得し，解放が終了するまでの一連の制御フロー F_1 が閉じている．このように，2 相ロックの場合も閉じた制御フローが得られる．P_3 の出したロック要求はロックが解放されるまで待たされるため，P_3 のロック要求の受信イベントの直後が中断イベントとなっており，制御フロー F_2 は P_1 がロックを解放した後の復帰イベントに続いている．イベントアブストラクションの結果は図 8 の (c) のようになる．

2 相ロックにおいて，バグのためにすでにロックされているデータに対して重複してロックしてしまった場合を考える（**図 9** の (a)）．この場合の制御フローを図 9 の (b) に示す．F_1，F_2 どちらの制御フローも閉じていることが分かる．このように，分散プログラム

にバグのある場合でも，閉じた制御フローが得られ，バグのない場合と同じ結果となることがある（図 9 の (c)）この場合，4.4.2 項のようにイベントアブストラクションの結果からただちにバグを特定することはできないが，ある抽象イベントに着目した際にそれとデータ依存関係のある抽象イベントが容易に分かるので，分散計算のログから着目する部分に影響しない部分を切り離すことができる．この例では，F_1 からなる抽象イベントに着目すると，F_2 に対応する抽象イベントとの間にデータ依存関係が存在することが分かる．これに基づいて，F_1 と F_2 に含まれるイベントをより詳細に見ていくことで，バグを検出することになる．

5. 可視化ツールの作成

提案手法を実際の分散計算の解析に適用し，その有効性を評価するため，提案したイベントアブストラクション手法により，分散計算の構造の簡略化を行い，結果を画面上にグラフとして表示するような可視化ツールを作成した．作成した可視化ツールに分散計算のログを入力として与えると，以下の 3 種類のグラフを表示することが可能である．

(1) 各イベントを頂点で表し，イベント間の制御依存関係，データ依存関係を有向辺で表したグラフ．
(2) (1) のグラフにおいて，閉じた制御フローごとに頂点を色分けしたグラフ．
(3) 閉じた制御フローを 1 つの抽象イベントに置き換え，内部の関係を隠蔽したグラフ．

5.1 実験結果

可視化ツールを実際に適用するために，分散プログラムを入力として，分散計算のログを出力するようなシミュレータを作成した．実験では，2 相コミットと 2 相ロックのプログラムを実装し，それぞれについて，バグのないものと，4.4.1 項，4.4.2 項で述べたようなバグを挿入したものを用いてシミュレーションを行い，結果を作成した可視化ツールに適用した．

プロセスの総数は 10 とし，2 相コミット（2PC）については他の 2 つのプロセスの同意をとるものをすべてのプロセスに 1 度実行させた．2 相ロック（2PL）については 10 のプロセスのうち 5 つをデータ管理プロセスとし，それらが管理しているある 1 つのデータのロックを要求するものを残りの 5 つのプロセスに 1 度実行させた．

それぞれの実行において，イベントアブストラクション前後のイベント数を比較した．実験結果を**表 1** に示す．2 相コミット，2 相ロックのいずれの場合も，イベント数が減少し分散計算全体の構造が大幅に簡略化されていることが分かる．またバグのないプログラムを入力として与えた場合，一連の処理が 1 つのイベントにアブストラクションされていることが確認できた．2 相コミットでバグのある場合，バグのない場合と比べて，アブストラクション後のイベント総数が増えている．これは，4.4.1 項で述べたように，プログラムのバグにより，制御フローが閉じない部分が生じたためである．このように，アブストラクションに失敗した部分に着目することにより，バグの発見が容易となる．一方 2 相ロックの場合は，アブストラクション後の結果はバグの有無にかかわらず同じであるが，この場合 4.4.2 項で述べたように，抽象イベント間のデータ依存関係からバグを発見するための手がかりが得られる．

表 1 実験結果
Table 1 Simulation results.

入力プログラム	適用前	適用後
2PC（バグなし）	160	10
2PC（バグあり）	154	19
2PL（バグなし）	60	10
2PL（バグあり）	60	10

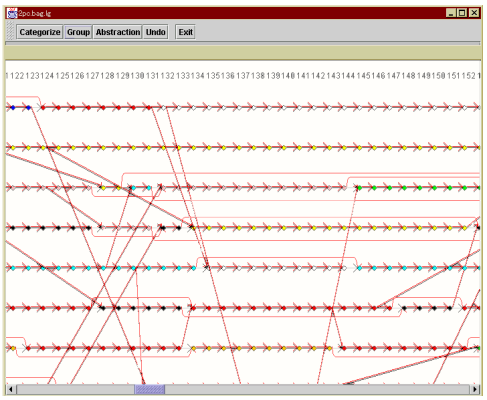

図 10 出力例（イベントアブストラクション前）
Fig. 10 An output of visualization tool (before event abstraction).

2 相コミットでバグのある場合の可視化ツールの出力例を**図 10** と**図 11** に示す．イベントアブストラクションによって，分散計算が大幅に簡略化されていることが分かる．

6. あとがき

イベントアブストラクションは，膨大なイベントからなる分散計算を簡略化し，その全体構造をおおまか

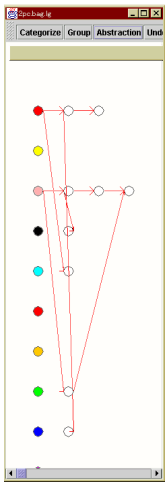

図 11 出力例（イベントアブストラクション後）
Fig. 11 An output of the visualization tool (after event abstraction).

に把握するために有効である．本研究では，従来のイベント間の因果関係を制御依存関係とデータ依存関係に分類し，制御依存関係より導かれる閉じた制御フローを 1 つの抽象イベントにアブストラクションする手法を提案した．この手法では，ログに記録する情報が少なくてすみ，また，あらかじめテンプレートを記述する必要もないため，解析者の負荷も軽減される．提案する手法を実装した可視化ツールにより，分散計算の構造の簡略化とその中に含まれるバグの発見に一定の有効性があることが確認できた．

参 考 文 献

1) Adelstein, F. and Singhal, M.: Real-time causal message ordering in multimedia systems, *Proc. 15th IEEE Intl. Conf. on Distributed Computing Systems*, pp.36–43, IEEE (1995).
2) Ahuja, M., Kshemkalyani, A.D. and Carlson, T.: A basic unit of computation in distributed systems, *IEEE Proc. 10th International Conference on Distributed Computing Systems*, pp.12–19 (1990).
3) Ahuja, M. and Mishra, S.: Units of computation in fault-tolerant distributed systems, *Proc. 14th IEEE Intl. Conf. on Distributed Computing Systems*, pp.626–633, IEEE (1994).
4) Alagar, S. and Venkatesan, S.: Causal ordering in distributed mobile systems, *IEEE Trans. Comput.*, Vol.46, No.3, pp.353–361 (1997).
5) Basten, T.: Breakpoints and time in distributed computations, *Proc. 8th International Workshop, WDAG '94*, Lecture Notes in Computer Science, Vol.857, pp.340–355 (1994).
6) Basten, T., Knuz, T., Black, J.P., Coffin, M.H. and Taylor, D.J.: Vector time and causality abstract events in distributed computations. *Distributed Computing*, Vol.11, No.1, pp.21–29 (1997).
7) Bates, P.C.: Debugging heterogeneous distributed systems using event-based models of behavior, *ACM Trans. Comput. Syst.*, Vol.13, No.1, pp.1–31 (1995).
8) Charron-Bost, B., Mattern, F. and Tel, G.: Synchronous, asynchronous and causally ordered communication, *Distributed Computing*, Vol.9, No.4, pp.173–191 (1996).
9) Jard, C. and Jéron, T.: A general approach to trace-checking in distributed computing systems, *Proc. 14th IEEE Intl. Conf. on Distributed Computing Systems*, pp.396–403, IEEE (1994).
10) Lamport, L.: Time, clocks and the ordering of events in a distributed system, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
11) Luckham, D.C., Kenney, J.J., Augstin, L.M., Vera, J., Bryan, D. and Mann, W.: Specification and analysis of system architecture using rapide, *IEEE Trans. Softw. Eng.*, Vol.21, No.4, pp.336–355 (1995).

(平成 12 年 5 月 22 日受付)
(平成 12 年 10 月 6 日採録)

多田 知正（正会員）

平成 5 年大阪大学基礎工学部情報工学科卒業．平成 7 年同大学大学院博士前期課程修了．平成 10 年同大学院博士後期課程退学．現在，同大学院基礎工学研究科助手．博士（工学）．分散システム，分散プログラミング環境に関する研究に従事．

樋口 昌宏（正会員）

昭和 58 年大阪大学基礎工学部情報工学科卒業．昭和 60 年同大学大学院博士前期課程修了．昭和 60〜平成 2 年（株）富士通研究所勤務．平成 3 年大阪大学基礎工学部情報工学科助手．平成 7 年同講師．現在，近畿大学理工学部電気工学科助教授．工学博士．通信プロトコル等の並行処理系の検証，試験，デバグに関する研究に従事．