フォールトのパターン化に基づく並行システムのデバッグ支援

張 漢明　　沢田 篤史　　野呂 昌満

南山大学理工学部

概 要

本研究の目的は，並行プログラムの同期問題に関するデバッグを支援することである．目的を達成するための基本的なアイデアは，プログラムのフォールトと軌跡の間の関係を定式化することである．本研究では，並行プログラムの典型的な同期問題に着目し，セマフォアを用いた同期問題の解法に対するプログラムのフォールトと軌跡の関係を，フォールトパターンとして提示することを目指す．本稿では，セマフォアを用いた第一種の読み書き問題の解法を対象として，フォールトと軌跡の間の定式化について議論する．

1　はじめに

並行プログラミングにおいて，システムの障害（failure）からプログラムのフォールト（fault）を特定するデバッグは困難な作業である．並行プログラムのデバッグでは，障害の状況から，障害に至る動作履歴を想定してプログラムのフォールトを特定する．並行プログラムではプロセスのコンテキストが非決定的に切り替わるので，検討すべき動作履歴は単一プロセスに比べて複雑になる．

McDowell らのサーベイ論文 [1] でまとめられている通り，並行プログラムのデバッグは古くから扱われている問題である．当時から現在に至るまでデバッグのための様々なツールやアプローチが継続的に提案されている [2, 3, 4, 5, 6, 7, 8, 9] ．これら様々なデバッグ支援技術に共通する特徴は，対象プログラムの実行時に観測されるログなどの情報を扱っていることである．実行時の情報を，それぞれの観点からのモデルに基づいて抽象化し，フォールト箇所の特定や修正に用いている．

並行プログラミングで肝心なことは，並行プロセス間での同期と通信であり，並行プロセス間における同期に関する典型的な同期問題（synchronization problem）がある [10]．同期問題には，「生産者-消費者問題」や「読み書き問題」などの問題があり，それぞれの問題に対する様々な解法が提案されている．同期問題の解法には同期基本命令が必要である．同期基本命令には，汎用性が高く言語に依存しないセマフォア（semaphore）がある．

本研究の目的は，並行プログラムの同期問題に関するデバッグを支援することである．並行プログラムにおける同期制御の基本構造は典型的な同期問題の解法にあると仮定して，同期問題の解法に対してプログラムの実行履歴から，プログラムのフォールトを特定する方法の提案を目指す．プログラムのフォールトとプログラム動作軌跡の間の関係が解明されれば，障害発生時の軌跡の情報からプログラムのフォールトへの対応付けが可能になると期待できる．

本研究の基本的なアイデアは，プログラムのフォールトと軌跡の関係を定式化して，プログラムの動作の軌跡からプログラムのフォールトへの対応づけを「フォールトパターン」として提示することである．プログラムのフォールトと軌跡の間の関係を図 1 に示す．フォールトと軌跡の間の関係は，フォールトの

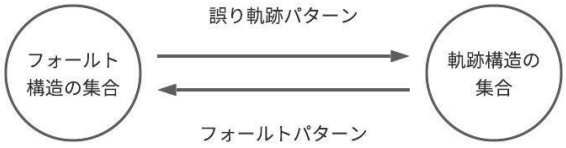

図 1: フォールトと軌跡の間の関係

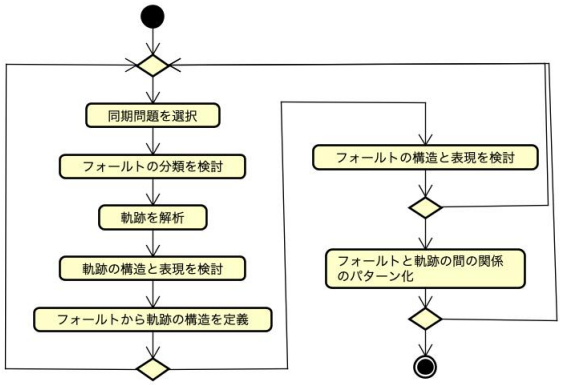

図 2: 研究の手順

構造と軌跡の構造の間で対応づけれらると想定した.「フォールト構造の集合」から「軌跡構造の集合」への写像を「誤り軌跡パターン」とし,「軌跡構造の集合」から「フォールト構造の集合」への写像を「フォールトパターン」と定義した.

本研究の研究課題は，典型的な同期問題の解法の領域で

プログラムのフォールトと軌跡の間の関係を定式化

することである．フォールトと軌跡の間の関係を定式化するための課題として次の 3 点があげられる.

- フォールトの分類
- フォールトを特定する軌跡の構造とその表現
- フォールトの構造とその表現

1 点目では，フォールトを分析する基準としてフォールトの分類を行う．フォールトの分類は，正しい解法に対するフォールト作成の基準の役割と，フォールトを表現するための手段としての役割がある．2 点目で，フォールトの分類が軌跡から検出できるフォールトを規定する．1 つのフォールトに対して軌跡は複数存在する．フォールトを判別可能な軌跡の構造を解明して，その表現方法を提示する．3 点目では，具体的なフォールトの記述を抽象化して，構造化されたフォールトの表現を用いて，フォールトと軌跡の間の関係を定式化することを目指す.

上記の研究課題を達成するために，以下の方法で研究を進める.

- セマフォアを用いた同期問題の解法 [10] を対象とする．
- 個々の具体的な問題に対してフォールトと軌跡の間の関係を定義する．
- フォールトと軌跡の間の関係をパターン化する．

具体的な研究の手順を図 2 に示す．図の左側の繰り返しは，具体的な同期問題を対象としてフォールトと軌跡の間の関係を解析する手順を示している．具体的な問題を通して，フォールトの分類と軌跡の表現を洗練する．その後，フォールトの表現を構造化して抽象的な表現を検討する．最後に，フォールトと軌跡の間の関係をパターン化する．

本稿では，セマフォアを用いた第一種読み書き問題の解法 [10] に対するフォールトパターンについて議論する．フォールトを判別するための軌跡の表記法を提案し，第一種読み書き問題の解法に対する単一のフォールトにおけるフォールトと軌跡の間の関係を定義する．複数のフォールトに対しては，単一のフォールトを 1 つずつ検出することにより，フォールトを減少することができる．

2 並行プログラムにおける同期問題

並行プログラミングで肝心なことは，並行プロセス間での同期と通信であり，並行プロセス間における同期に関する典型的な同期問題（synchronization problem）がある．文献 [10] には，同期問題として「相互排除問題」「生産者-消費者問題」「眠り床問題」「読み書き問題」「五人の哲学者問題」「喫煙者問題」を取り上げ，セマフォアを用いた解法が示されている．

セマフォアは（semaphore）は値を持ち，P 命令もしく V 命令を用いてプロセスの同期を制御する．セマフォアの値を 0 と 1（あるいは false と true）に限ったとき 2 値セマフォアという．整数の値としてとるセマフォアを整数セマフォアという．セマフォアは初期値を設定して利用する．

3 フォールトパターン

本節では，対象とする並行システム，軌跡，フォールトと軌跡の関係，および，パターンについて説明する．

3.1 対象とする並行システム

本研究で対象とする並行システムの構造を定義する．

並行システム

並行システムは 3 項組 (PS, V, E) として定義する．PS はプロセスの集合である．並行システムは複数のプロセスがインターリーブにより並行合成されたものとみなす．V はプロセス間の変数の集合である．変数は全てのプロセスから参照できる共有変数である．変数の内部構造はここでは規定していない．変数は代入により値が変化しその値は環境で保持される．環境 E は変数の集合 V から値の集合への写像である．

プロセス

プロセスにはプログラムが対応づいている．プログラムは，アクションの列として定義する．

アクション

アクションはプログラムの基本構成要素の集合である．基本構成要素は，命令型プログラムの基本構成要素にセマフォアの同期命令を加えたものである．命令型プログラムの基本構成要素は，代入，条件，繰り返しとする．セマフォアの同期命令は $P(x)$ と $V(x)$ とする．x はセマフォアを区別するための名前である．

3.2 軌跡

上記で定義した並行システムに対して，その動作履歴を表す軌跡に関する用語を定義する．

システム S の軌跡集合

軌跡集合 $Tr(S)$ は並行システム S の軌跡の集合として定義する．軌跡は S が実行時に起こりうる動作履歴を表す．$Tr(S)$ は軌跡として実行の停止前の全ての動作履歴を含む．

軌跡

軌跡は，実行の有限列として定義する．実行は 3 項組 (P, A, E) である．P はプロセス，A はアクション，E は環境である．

3.3 フォールトと軌跡の関係

フォールトと軌跡の関係に関連する概念を定義する．

誤り軌跡

誤り軌跡は，正しいシステムに対して，フォールトの存在を判定できる軌跡とする．

誤り軌跡集合

誤り軌跡集合は，正しいシステムに対して，フォールトの存在が判定できる軌跡の集合である．正しいシステム cs に対する誤り軌跡集合は，以下の写像 ETr_{cs} で定義される．

$$ETr_{cs}(fs) = Tr(fs) - Tr(cs)$$

fs はフォールトを含んだシステムである．誤り軌跡集合は，図 3 で示された灰色の部分で，フォールトを含んだシステム fs の軌跡集合と正しいシステム cs の軌跡集合の差である．

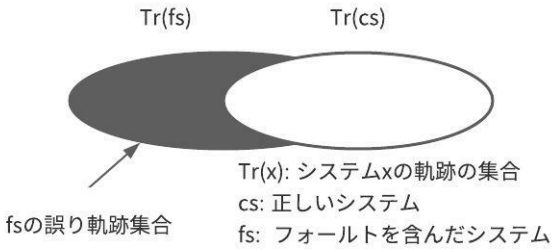

図 3: 誤り軌跡集合

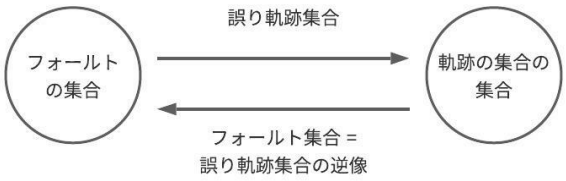

図 4: フォールト集合

フォールト集合

フォールト集合は，軌跡からフォールトへの対応とする．フォールト集合は，図 4 に示すように，誤り軌跡集合の逆像として定義できる．フォールトと軌跡の関係は，誤り軌跡集合を定義することで一意に得ることができる．

3.4 パターン

個々の解法に対するフォールトと軌跡の間の関係を，抽象化した構造をパターンとして定義する．

誤り軌跡パターン

誤り軌跡パターンは，図 1 に示すように，フォールトの構造から誤り軌跡の構造への写像である．

フォールトパターン

フォールトパターンは，誤り軌跡の構造からフォールトの構造への逆像を考えることで定義することができる．

4 フォールトに対する軌跡の分析

典型的な同期問題の 1 つである第一種の読み書き問題を対象として，プログラムのフォールトに対する誤り軌跡を解析する．セマフォアを用いた第一種の読み書き問題の解法 [10] に対してフォールトを分類して，それぞれのフォールトに対する誤り軌跡を検討する．誤り軌跡を簡潔に表現するために，正規表現の拡張を定義して，フォールト集合について検討する．

4.1 フォールトの分類

同期問題のプログラムは，P 命令と V 命令の使い方と，共有変数の使い方の誤りの 2 種類に分類される．それぞれの単一のフォールトを検討した後で，複数のフォールトについての誤り軌跡を検討する．

4.1.1 セマフォアの誤用

セマフォアの同期命令の間違った使い方について検討する．同期命令には P 命令と V 命令がある．セマフォアは複数存在する場合があるので，セマフォアを識別するために名前がある．同期命令は，名前を含めて P(名前)，V(名前）と表現する．セマフォアの誤った使い方として以下の 1 から 4 が考えられる．

1. セマフォア初期値誤り
2. セマフォア抜け
3. セマフォア名前誤り
4. P-V 命令取り違い

1 について．セマフォアの初期値の値を間違えた誤りを「セマフォアの初期値誤り」とする．セマフォアを利用する場合，その初期値の設定が必要である．相互排除を実現するためには，セマフォア s の初期値を 1 として

```
P(s); きわどい区間 V(s);
```

とすれば良い．整数セマフォアの場合は，初期値として整数値が用いられる．セマフォアの初期値誤りはこの初期値を間違えた場合である．

2 について，何らかの要因でセマフォアが抜け落ちている誤りを「セマフォア抜け」とする．

3 について，セマフォアの名前の指定を間違えた誤りを「セマフォア名前誤り」とする．同期問題の解法には，複数のセマフォアが用いられる場合がある．セマフォア名前誤りは，正しいセマフォアの名前を間違えた場合の誤りである．例えば，セマフォア s の P 命令である P(s) とすべきところを，名前を間違えた P(t) とした場合である．

4 について，セマフォアの P 命令と V 命令を間違えた誤りを「P-V 命令取り違い」とする．例えば，セマフォア s の P 命令である P(s) とすべきところを，V(s) と間違えた場合である．

4.1.2 共有変数の誤用

同期問題の制御に用いられる共有変数に関する誤りについて検討する．共有変数の誤った使い方として以下の 1 から 4 が考えられる．

1. 変数初期値誤り
2. 変数更新誤り
3. 変数名誤り
4. 比較誤り

変数の初期値を間違えた誤りを「1. 変数初期値誤り」とする．共有変数で用いられる変数は，通常，初期値が設定されている．変数初期値誤りは，この変数の初期値を間違えた場合である．

共有変数の代入時における更新する値の式の誤りを「2. 変数更新誤り」とする．変数更新誤りは，同期問題に関わる変数の更新値を間違えた場合である．

共有変数の変数名を間違えた誤りを「3. 変数名誤り」とする．同期問題の解法で複数の変数が用いられた場合，例えば，変数として a と b が用いられた場合，a の変数を使用するところを誤って b の変数を使用した場合である．

条件の記述にを間違えた誤りを「4. 比較誤り」とする．C 言語などのプログラミング言語では，条件式で代入を使用することができる．条件式に代入が用いられた場合の誤りは，比較誤りの例である．

4.2 誤り軌跡の検討

セマフォアを用いた第一種の読み書き問題の解法を対象として，前節で示したフォールトの分類に基づいて誤り軌跡を検討する．

4.2.1 第一種の読み書き問題とその解法

同一の資源をアクセスする読みプロセスと書きプロセスがあり，読みプロセスは同時にアクセスできるが，書きプロセスは排他的にアクセスしなければならない場合がある．第一種の読み書き問題では，読みプロセスは，書きプロセスがアクセスしていない限り，資源にアクセスできる，書きプロセスは，読みプロセスがいっさい資源をアクセスしていないときに限り，資源にアクセスできる，文献 [10] で紹介されている第一種の読み書き問題の解法を図 5 に示す．以降では，読みのプロセスを R，書きのプロセスを W とする．

4.2.2 セマフォアの誤用

第一種の読み書き問題において，4.1.1 節で示したセマフォアの誤用の 1 から 4 は，具体的には以下の誤りに対応する．

セマフォアの初期値誤り (1) セマフォアの初期値が誤っていた場合は，最初の P 命令もしくは V 命令実行後のセマフォアの値が正しいプログラムと違った値になる．図 5 の解法には，プロセス R と W の両方とも位置 01 に P 命令がある．セマフォアの初期値が 1 の場合，最初の P 命令の実行はセマフォアの値を 0 にする．誤り軌跡は，この P 命令の実行後のセマフォアの値が 0 以外の値となる．

```
読みのプロセス (R)      書きのプロセス (W)
01:P(m);                01:P(w);
02:   rc = rc + 1;      02:  write;
03:  if(rc == 1)        03:V(w);
04:     P(w);
05:V(m);
06:  read  ;            共有変数
07:P(m);                int rc = 0;
08:   rc = rc -1;
09:   if(rc == 0)       セマフォア
10:      V(w);          m, w: 初期値 1
11:V(m);
```

図 5: 第一種の読み書き問題の解法

セマフォア抜け (2)　セマフォアの命令が何らかの原因で実行されないことは，その命令の前後の実行履歴から特徴づけることができる．例えば，プロセス R の位置 04 の P(w) の抜けは，実行の履歴が

- 位置 03 における rc の値が 1,
- 位置 05 の V(m)

から特定することができる．

セマフォア名前誤り (3)　セマフォアの名前の誤った使用は，その誤りの実行が軌跡に含まれる．例えば，プロセス R の位置 04 の P(w) から P(m) への間違いは，実行の履歴が

- 位置 03 で rc の値が 1,
- 位置 04 の P(m)

から特定することができる．

P-V 命令取り違い (4)　P 命令と V 命令の使用を取り違えた誤りは，セマフォア名前誤りと同様の誤り軌跡となる．例えば，プロセス R の位置 04 の P(w) を V(w) に間違えた場合は，実行の履歴が

- 位置 03 で rc の値が 1,
- 位置 04 の V(w)

から特定することができる．

4.2.3　共有変数の誤用

第一種の読み書き問題において，4.1.2 節で示したセマフォアの誤用の 1 から 4 は，具体的には以下の誤りに対応する．

変数初期値誤り (1) 変数の初期値が誤っていた場合は，最初に変数を利用する実行で誤りを検出することができる．図 5 の解法では変数が rc である．変数 rc の最初の実行は，プロセス R の位置 02 である．変数の初期値が間違っていた場合，位置 02 の実行後の変数の値が 1 以外の値になる．

変数更新誤り (2) 変数の更新値を誤った場合は，誤った更新の最初の実行でフォールトを検出することができる．プロセス R の位置 02 の更新値を間違えた場合，その間違えたアクションが軌跡に記録される．

変数名誤り (3) 変数名を誤って使用した場合は，上記の変数更新誤りと同様にして誤りを検出することができる．

比較誤り (4) 条件文で，比較の式を誤った場合は，上記の変数更新誤りと同様にして誤りを検出することができる．

4.2.4 単一フォールトの誤り軌跡の特徴

単一フォールトの誤り軌跡の特徴を以下に示す．

- 誤りがあるプロセスに制限した軌跡上で検出することができる．
- 最初に「誤りを実行」したプロセスで検出することができる．「誤りを実行」とは，混入した誤りの箇所を実行することである．
- 誤りの検出に複数の実行が必要な場合がある．抜けの検出や条件文が関係する場合は，1 つのアクションで検出することができない．継続したアクションの履歴から判別することができる．

4.2.5 複数フォールト

複数のフォールトが存在した場合の誤り軌跡について検討する．複数のフォールトについて，次の 2 つに場合分けして考える．

1. 同一プロセス内における複数のフォールトと
2. 複数プロセスにおける複数のフォールト

同一プロセス内における複数フォールト (1) 同一プロセス内における複数のフォールトがある場合は，複数のフォールトの中でフォールトの位置が最初の位置のフォールトを検出することができる．ただし，フォールトの特定がフォールトの位置以降の実行に依存する場合，別のフォールトによる履歴の影響を考慮する必要がある．次の位置のフォールトは，最初の位置のフォールトを取り除くことにより検出することができる．

複数プロセスにおける複数フォールト (2) 複数プロセスにおいて複数のフォールトがある場合には，各プロセス毎に上記の同一プロセス内における複数のフォールトを適用して，プロセス毎に最初のフォールトを検出することができる．同一プロセス内における複数フォールトと同様に，最初のフォールトを取り除くことにより，次の位置のフォールトを検出することが可能になる．

4.3 軌跡のパターン表現

第 4.2 節の分析結果に基づいて，軌跡のパターン表現を検討する．誤り軌跡を簡潔に表現するために正規表現の拡張表記を検討する．

4.3.1 実行の表現

軌跡は実行 (P,A,E) の列で定義されている．P はプロセス，A はアクション，V は環境である．実行を以下のように表現することにする．

（プロセス名，アクション，変数名と値の組の列）

プロセスはプロセス名，アクションは基本構成要素，変数名と値の組はアクション実行後の環境の変数とその値である．例えば，(R,rc=rc+1,(rc,1)) は，プロセス R がアクション rc=rc+1 を実行した後で，変数 rc の値が 1 であることを表している．

4.3.2 誤り軌跡のパターン表現

第 4.2.4 節における分析から，誤り軌跡を表現するための特徴として，次の 2 つがあげられる．

1. あるプロセスに着目した実行列
2. ある実行位置からの継続したアクションの実行

これらの特徴を簡潔に表すための表現について考える．

着目するプロセスに制限した軌跡 (1) 誤り軌跡は，ある着目したプロセスの軌跡に対して特徴付けられる．システムの軌跡 tr に対して，あるプロセス P に制限した軌跡を P ↑ tr で表現する．この場合，実行の表現を (P,A,E) からプロセス P を省略することが可能となる．

実行位置による軌跡の表現 (2) 誤り軌跡は，正しい軌跡の後で複数の実行により特徴付けられる．軌跡の表現を，誤り軌跡を特徴づける最初の実行位置から表現することにする．この実行位置を，実行の項目に追加することにする．

誤り軌跡のパターン表現

以上の議論から，誤り軌跡を

P ↑ tr(L,A,(V,Val) の列) の列

で表現することにする．P はプロセス名，tr は軌跡，L は実行位置，A はアクションの基本構成要素，V は変数名，Val は値である．

上記の誤り軌跡の表現に加えて，正規表現の拡張として以下のメタ文字を導入する．

- .：任意の行，アクション，値などにマッチ
- ^X：X 以外にマッチ

4.4 第一種の読み書き問題の誤り軌跡

図 5 の解法に対して，第 4.2 節で分析した誤り軌跡を，上記の誤り軌跡のパターン表現で記述する．誤り軌跡のパターン表現の例を以下に示す．

4.4.1 誤り軌跡のパターン表現の例

第一種の読み書き問題に対する誤り軌跡のパターン表現の例を以下の (1) から (3) に示す．

(1) R のセマフォア m の初期値誤り

- `R ↑ tr(1,P(m),(m,^0))`

上記の表現は，プロセス R に制限した軌跡上で，実行位置 1 のアクションが `P(m)` で，実行後の変数 `m` の値が 0 以外の軌跡の集合を表す．

(2) R の位置 04 のセマフォア名前誤り

- `R ↑ tr(4,P(^w),.)`

上記の表現は，プロセス R に制限した軌跡上で，実行位置 4 のアクションが P(w) 以外である軌跡の集合を表す．

(3) R の位置 10 のセマフォア V(w) 名前誤り

- `R ↑ tr(.,if(rc==0),(rc,0))(.,V(^w),.)`

この誤りは，位置 09 での rc の値が 0 のときに検出することができる．rc の値は違うプロセスが更新する場合があるので，この誤りはいつ実行されるかわからない．上記の表現は，プロセス R に制限した軌跡上で，`if(rc==1)` での変数 `rc` の値が `0` で，その次の実行が `V(w)` 以外のアクションである軌跡の集合を表す．

4.4.2 誤り軌跡集合とフォールト集合

第一種の読み書き問題における単一フォールトに対する誤り軌跡集合のパターン表現が定義できた．セマフォアの誤用で 27 パターン，共有変数の誤用で 7 パターンの誤り軌跡を定義した．個々のフォールトに対する誤り軌跡集合に重なりはなかった．軌跡からフォールトへの対応であるフォールト写像は逆写像として定義することができる．つまり，軌跡のパターンを発見すれば，フォールトを一意に特定することができる．

5 考察

本節では，検出されないフォールト，フォールトおよびアクションの抽象化，既存研究との比較を考察する．

5.1 検出されないフォールト

本稿で示したフォールト集合で検出できないフォールトについて考える．検出できないフォールトには，実行時に誤りが実行されない場合と，フォールトの分類に漏れがある場合がある．

(1) 誤りの未実行 条件文の本体に誤りがある場合，プログラムの実行で条件文の本体が実行されない場合がある．例えば，図 5 の解法の R の位置 10 のセマフォア V(w) の名前誤りについて考える．位置 10 は，位置 09 において変数 `rc` の値が 0 のときに実行される．もし，読みプロセスが読み続けて書きプロセスが実行されない状況のとき，位置 09 で `rc` が 0 になることがない．このとき，誤りが実行されないので軌跡にも誤りが記録されない．しかし，これは正しい解法でも起こりうる状況である．このような場合は，実行カバレジなどの情報を用いて，別の方法を検討する必要がある．

(2) フォールト分類の漏れ 誤り軌跡集合の定義は，フォールトの分類を基準としてフォールトを作成したので，フォールトの分類以外のフォールトを軌跡から検出することができない．例えば，共有変数の誤用では，共有変数抜け，という誤りがない．誤り分類は恣意的なので，誤りの分類を完全に行うことはできない．第一種の読み書き問題以外の問題を解析することにより，より網羅的な誤り分類を作成することが期待できる．

5.2 フォールトの抽象化

フォールトの表現について，具体的なソースコードによる表現を抽象化することを考える．フォールトの表現を抽象化するためには，解法を構造化した表現にする必要がある．セマフォアを用いた同期問題の解法では，P-V 命令に着目した構造化が考えられる．

例えば，図 5 の解法では，読みプロセスは相互排除の繰り返しとみなすことができる．具体的には，P(m) と V(m) の組が (1) 位置 01-05 と (2) 位置 (07-11）にある．誤り軌跡は (1) の P-V 命令の間に存在する場合と，(2) の P-V 命令の間に存在する場合がある．(1) の P-V 命令は，必ず最初に実行するプロセスがあるので，この位置にある誤りは，その最初のプロセスが実行する特定の実行位置でフォルトを検出できる．(2) の P-V 命令の間に存在する誤りについては，実行の状況によって動作が変わるので，特定の固定位置でフォールトを検出することができない場合がある．P-V 命令の制御構造と P-V 内の制御構造を分析して，フォールトの構造と軌跡の構造の関係を定式化することが今後の課題である．

5.3 アクションの抽象化

同期問題の解法の表現を，具体的なプログラミング言語に依存しない表現に抽象化することにより，フォールトパターンの汎用性を高めることができる．本稿では，具体的なプログラミング言語を用いた同期問題の解法に対して，フォールトと軌跡の関係を解析した．図 5 の解法において，例えば，`rc=rc+1` を「読み手を増やす」，`rc==1` を「最初の読み手」のように，プログラミング言語に依存しない表現するすることができる．アクションの記述を抽象化することにより，プログラムの実現方法を与えることで，具体的なフォールトパターンを導出することが可能になる．

5.4 既存研究との比較

並行性を持つプログラムのデバッグは古くから扱われている問題である．McDowell ら [1] は並行プログラムのデバッグにおける問題点と，デバッグに向けた基本的なアプローチを分類し，様々な研究成果や商用ツールについて紹介している．中でも多くの技術は，デバッガを用いたブレークポイントの設定と実行経路の解析に基づくとしている．

Tzoef ら [2] は，複数の製造制御ソフトウェアを対象に，プログラム上の誤りの恐れのある箇所を特定するアルゴリズムを提案している．プログラムの計測ポイントが実行されたトレースログに機械学習技術を用いている点に特徴がある．

Fleming ら [3] は，学生を被験者とした並行プログラムの修正実験を通じ，並行プログラムのデバッグにおける戦略の分類を行っている．デバッグに成功する学生の傾向として，並行プログラムの典型的なスタイルに着目していること，実行トレースを抽象化した振舞いモデルを作成していること，を挙げている．

Yuan ら [4] は，製品の故障ログとプログラムのソースコードからフォールト箇所と原因の抽出を支援するツール SherLog を提案し，実システムへの適用を通じてその有効性を評価している．ソースコードの静的解析結果とログ情報を紐づけすることで，デバッグに有用な情報を提示することができる点を特徴とする．

Yuan ら [5] は，並行システムを含む大規模システムのフォールト箇所や原因を特定するために，システムログ上の情報を拡張するツール LogEnhancer を提案している．文献 [4] のアプローチと同様，ソースコードの静的解析結果とログ情報と紐づけて利用していることに特徴がある．

Volos ら [6] は，マルチスレッドプログラムの同期ミスに起因するデッドロックなどの故障を回避するため技術について検討と実証的調査を行っている．プログラムの並行性を高め，実行効率を向上させるために，プログラム上のきわどい領域を宣言する Transactional Memory（TM）と呼ぶ仕組みがある．この研究では，実用に供されている大規模なシステムにおいて TM がどのように用いられ，故障回避に役立っているかを調査した．

Park[7] は，デッドロックを生じない並行プログラムのバグを発見，特定するための情報提供ツールを提案している．フォールト箇所の特定のために必要な情報をアクセスログからパターン化して抽出し，利用者に提示する点が特徴となっている．

Yu ら [8] は，システムログのみを用いて，並行プログラムの故障を自動的に再現するツール DESCRY を提案している．DESCRY は，ソースコードの静的解析と記号実行技術を併用し，トレースログと情報統合することで，再現が困難なタイミングに起因する並行システムの故障の再現に成功している．

Bianchi ら [9] は，クラッシュしたプログラムのスタック情報から，並行プログラムおけるスレッド安全性に違反する故障を再現するためのテストケースを自動生成するツール ConCrash を提案している．テストケースの生成は，情報探索手法とインタリーブするスレッドの実行トレースを分離するための技術を組み合わせることで実現している．

Huang ら [11] は，並行性に起因する不具合を再現可能とするために，スレッド別の実行ログを記録する方法 Clap を提案している．実行ログをスレッドごとに分別することで，効率の良い分析と再現を可能としている．

Farchi ら [12] は，並行性に起因する不具合の体系的な分類に基づく並行バグパターン（Concurrent Bug Pattern）を提案している．提案するバグパターンと並行デザインパターンとの対応づけにより，設計上の誤りと不具合との関係の定式化を試みている．

Huang ら [13] は，並行プログラムのログ解析における予測的不具合発見手法の効率化に取り組んでいる．予測的不具合発見手法とは，実行ログから今後起こりうる不具合を推測するものであり，大規模な並

行プログラムへの適用が望まれている．この研究では実行ログから冗長な情報を削除し，抽象化することで，大規模な並行プログラムの分析を可能とする方法を提案している．

これらの文献などで扱われているデバッグ支援技術に共通する特徴は，

- アクセスログやトレースログ，メモリダンプを利用し，フォールトの特定や修正に必要な情報を抽出
- 抽出した情報を何らかのモデルに基づいて抽象化し，デバッグを実施する開発者にわかりやすく提示

である．本研究では，正しいプログラムの解法が存在することを前提として，プログラムの実行軌跡からプログラムの詳細な欠陥を特定することを目指していることが，上記の既存研究との相違点である．

6 おわりに

本稿では，並行プログラムにおける同期問題を対象として，軌跡からフォールトを特定するために，

- フォールトの分類
- フォールトを特定するための誤り軌跡のパターン表現

を提示した．P-V 命令を用いた第一種の読み書き問題の解法に対するフォールト集合を示し，フォールトと軌跡の間の関係の定式化の見込みを示した．

今後の課題として，

- 他の同期問題に適用
- フォールト表現の構造化

があげられる．文献 [10] にある典型的な同期問題における P-V 命令を用いた解法に対して，誤り軌跡集合を定義して，誤り軌跡のパターン表現を洗練化する．同期問題の制御は，P-V 命令の構造に依存しているので，それらの基本構造を解明して，プログラムの基本命令と関連させることにより，フォールト表現の構造化が可能になると考えられる．

本稿で提案するフォールトパターンの応用先として，IoT（Internet of Things）[14] などの CPS（Cyber Physical Systems）アプリケーションが挙げられる．著者らがこれまでに提案してきたアーキテクチャ [15] に基づく並行プログラムに対する本手法の有効性を確認するなど，現実的なアプリケーションに対する適用についても今後の課題としたい．

謝辞

本稿は，南山大学大学院理工学研究科ソフトウェア工学専攻博士前期課程の修士論文に著者らが修正を加え，まとめたものである．当該修士論文の研究指導にあたり，故青山幹雄先生には厳しくも有益なコメントを数多くいただいた．ここに青山先生への感謝を申し上げるとともにご冥福をお祈りする．

参考文献

[1] Charles E. McDowell and David P. Helmbold, “Debugging Concurrent Programs”, ACM Computing Surveys, Vol. 21, No. 4, pp. 593–622, 1989.

[2] Rachel Tzoref, Sumuel Ur and Elad Yom-Tov, “Instrumenting Where it Hurts— An Automatic Concurrent Debugging Technique”, Proc. ISSTA’07, ACM, pp. 27–37, 2007.

[3] Scott D. Fleming, Eileen Kraemer, R. E. K. Stirewalt, Shaohua Xie and Laura K. Dillon, “A Study of Student Strategies for the Corrective Maintenance of Concurrent Software”, Proc. ICSE’08, ACM, pp. 759–768, 2008.

[4] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou and Shankar Pasupathy, “SherLog: Error Diagnosis by Connecting Clues from Run-time Logs”, Proc. ASPLOPS’10, ACM, pp. 143–154, 2010.

[5] Ding Yuan, Jing Zheng, Soyen Park, Yuanyuan Zhou and Stefan Savage, “Improving Software Diagnosability via Log Enhancement”, Proc. ASPLOPS’11, ACM, pp. 3–14, 2011.

[6] Haris Volos, Andres Juan Tack, Michael M. Swift and Shan Lu, “Applying Transactional Memory to Concurrency Bugs”, Proc. ASPLOPS’12, ACM, pp. 211–222, 2012.

[7] Sangmin Park, “Debugging Non-deadlock Concurrency Bugs”, Proc. ISSTA’13, ACM, pp. 358–361, 2013.

[8] Tingting Yu, Tarannum S. Zaman and Chao Wang, “DESCRY: Reproducing System-Level Concurrency Failues”, Proc. ESEC/FSE’17, ACM, pp. 694–704, 2018.

[9] Francesco A. Bianchi, Mauro Pezzè and Valerio Terragni, “Reproducing Concurrency Failures from Crash Stacks”, Proc. ESEC/FSE’17, ACM, pp. 705–716, 2017.

[10] 土居範久, 相互排除問題, 岩波書店, 2011.

[11] Jeff Huang, Charles Zheng and Julian Dolby, “CLAP: Recording Local Executions to Reproduce Concurrency Failures”, Proc. PLDI’13, ACM, pp. 141–151, 2013.

[12] Eitan Farchi, Yarden Nir and Shmuel Ur, “Concurrent Bug Patterns and How to Test Them”, Proc. IPDPS’03, IEEE, 2003.

[13] Jeff Huang, Jinguo Zhou and Charles Zheng, “Scaling Predictive Analysis of Concurrent Programs by Removing Trace Redundancy”, ACM Transactions on Software Engineering and Methodology, Vol. 22, No. 1, pp. 1–21, 2013.

[14] Alessandro, M., BauerMartin, F., KrampRob, van, K., Sebastian, L., Stefan, M., ”Enabling Things to Talk”, Springer, 2013.

[15] 横山史明，沢田篤史，野呂昌満，江坂篤侍，“IoT の柔軟な相互運用性を実現するソフトウェアアーキテクチャの提案”，情報処理学会論文誌，Vol. 62，No. 4，pp. 995–1007，2021.