May 31, 2010

RT0909

Computer Science 17 pages

Research Report

Method to Extract Control Structures of Conditional Branch and Iteration for Generating Sequence Diagrams from Execution Traces

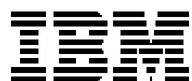
Satoshi HARAGUCHI and Kouichi ONO

IBM Research - Tokyo

IBM Japan, Ltd.

1623-14 Shimotsuruma, Yamato

Kanagawa 242-8502, Japan



1　目的

既存のレガシーシステムの振舞を分析するためのモデルを作成するリバースエンジニアリング技術である。振舞のモデル作成の目的としてはパフォーマンス分析を想定する。レガシーシステムの動的情報 (実行トレース) を観測し、それを元に、条件分岐や反復の制御構造を持つ全般化したシーケンス図を生成する。このような全般化された構造化シーケンス図は、振舞の理解と、システムの振舞の検証に利用することができる。これを人手のみで作業するのは多大な人的コストを要する。この作業を支援する機械的手段を提供することが目的である。

2　既存技術とその課題

システムの動的情報からモデルを得る既存のリバースエンジニアリング技術は大別して、関数呼出系列のみを入力とする技術と、それに加えて条件分岐や反復の制御情報も入力とする技術がある。後者は制御情報を得ることでソースコードの制御構造と合致する構造化シーケンス図を得ることができる代わり、実行観測の侵襲性を悪化させる。前者は後者より低侵襲である代わりに関数呼出系列から制御構造を推測・発見する必要がある。パフォーマンス分析の目的のためには、観測行為自体のシステムパフォーマンスへの影響が低い、すなわち低侵襲であることが条件となる。このため、前者の技術が必要となる。

前者の既存技術としては、パターンマッチングによって制御構造を発見してイベントチャートに変換する手法がある (Jerding 他、ICSE ’97)。パターン定義は heuristics に基づいており、したがって ad hoc な変換にならざるを得ない問題がある。この他に、関数呼出系列からε-NFA を作成してε-遷移と状態を除去した上で正規表現に変換して構造化シーケンス図を生成する手法がある (中村他、JP8-2008-0082、出願済み)。しかしこの方法は計算量が指数的になり、かつ、得られる制御構造が、望んでいる構造になることが保証できない問題がある。

3　技術の概要

実行トレース間および実行トレース内の関数呼出 (部分) 列のマッチングを取り、その際に、呼出列の編集距離 (レーベンシュタイン距離) を元に、望ましい制御構造を判断する。関数呼出の部分列の切り出しとマッチングは一種の Dynamic Programming となるが、編集距離が閾値を越える時点で探索を打ち切るため、計算量は抑えられる。

部分列の切り出しは、それまでの探索で得た部分列の群に対する編集距離が極小となる位置でおこなうため、共通部分と差異のある部分が明確に区分され、視覚的に望ましい制御構造を得ることが期待できる。またマッチング規則はあらかじめ複数の規則を定義してあり、対象システムの実装コードの特性に応じて選択・組み合わせることで柔軟に対応することができる。

4　当該技術のカバレージ・代替技術との差別化要因

代替技術としては既存のパターンマッチング手法があるが、heuristics に基づいた ad hoc な変換であるため、適用可能性および生成したモデルが望んでいる制御構造を持つかどうかはパターンをどのように定義したかに依存することになる。

この技術は、関数呼出部分列の切り出しは編集距離に基づいて探索するように定めてあるため、ad hoc な手法の問題を回避している。一方で、heuristics が必要とされるのはマッチングの処理であり、これについてはあらかじめ定義した複数のマッチング規則から、対象システムの実装コードの特性に応じて選択・組み合わせることを可能にしており、既存のパターンマッチングによる技術と同程度の柔軟さを損ねないようにしてある。

以上のことから、この技術は既存の技術に対して優位であり、より適切なモデルを得るにはこの技術を適用しない限り達成できない。

5　既存技術とその課題

プログラム実行時の動的情報 (実行トレースなど) からシーケンス図やイベントトレース図などを求めるリバースエンジニアリング技術において、条件分岐や反復などの制御構造を視覚化する技術は以下のものがある。

文献 [1] では、条件分岐や反復の制御情報もプログラム実行時に取得し、メタモデルに対応して変換する方法を提案している。制御情報まで取得することは侵襲性を悪化させるため、低侵襲な挙動観測により関数呼出系列のみを実行トレースとして取得する場合には変換できない。

文献 [2] では、パターンマッチングによって条件分岐や反復を発見する方法を提案している。パターンは heuristics で定義するため、ad hoc な変換となる。

文献 [3] は IBM Research による Jinsight についての論文である。この技術は、メソッドコールの反復を発見するのみなので、制御構造を発見することはできない。

文献 [4] では、実行トレースからε-NFA を作成し、ε-遷移の除去および状態の除去によって、状態数を削減した上で正規表現に変換し、UML 2.0 の構造化シーケンス図に変換する方法を提案している。ε-遷移の除去の計算量は状態数 n に対して O(n^3)、状態を除去した上での正規表現への変換には全体で O($n^4 * 4^n$) となる上に、得られたシーケンス図が望んでいる制御構造を持つ保証がない。

この文書で提案する技術は、以下の課題を解くことを目的としている。

1. 関数呼出系列のみからなる実行トレースから構造化シーケンス図に変換する (Briand らの手法のように制御情報を必要としない)
2. 指数的な計算量にしない (中村らの手法ではε-NFA からε-遷移と状態除去の上で正規表現に変換する計算量は指数的になる)

3. 望ましい制御構造を得る (中村らの手法では呼出の差異をループ内に積極的に取り込まないので煩雑で理解しにくい)

6 手法の詳細

6.1 定義

6.1.1 ログ

対象プログラムを実行して得られる一連の関数呼出記録を以下のように構造化したものと定義する（main 関数などのエントリーポイントは含めない)。

1. ある関数 A 内で、関数 B が呼び出されるとき、改行・インデントして記述する
   A
       B
2. ある関数の中で、関数 A , 関数 B が順次呼び出されるとき、改行して記述する
   A
   B

6.1.2 関数 A の階層

関数 A における、エントリーポイントからの呼出深度と定義する。関数 A の階層が d のとき、「A は第 d 層である」ともいう。関数 A 以下の階層の関数呼出の集まりも、関数 A が第 0 層に位置するログとなる。

以下のログ s があるとする。

```
A
    B
        C
D
    E
    E
```

このログ s は、関数 A 内で関数 B が呼ばれ、関数 B 内で関数 C が呼ばれ、関数 A の処理が終わると関数 D が呼ばれ、関数 D 内で関数 E が 2 回呼ばれたものである。

個々の関数呼出の階層は以下の通りである。

- 関数 main の階層 = 0
- 関数 A の階層 = 1
- 関数 B の階層 = 2
- 関数 C の階層 = 3

- 関数 D の階層 = 1
- 関数 E の階層 = 2

6.1.3 ログ s における関数呼出パス

ログにおける特定の関数呼出の指定は、階層 0 からその関数に至るパスとなる。前述の例において、第 1 層の関数 A 中の関数 B の関数呼出パスは /A/B (第 0 層の関数呼出 main は省略) 第 1 層の関数 D 中の 1 番目の関数 E の関数呼出パスは /D/E[1] (1 から開始)、となる。

6.1.4 部分ログ

ログ中のある関数以下の関数呼出の集まりを部分ログと呼ぶ。部分ログもログの性質を満たす。

6.1.5 ログ s における関数呼出パス p に対応する部分ログ

ログ s と関数呼出パス p が与えられたときに、p に対応する関数呼出を第 0 層とする部分ログを、p に対応する部分ログと呼ぶ。

6.1.6 ログ s の直下呼出階層

あるログにおいて、その第 0 層の関数から直接に呼び出される (第 1 層の) 関数の列のことを、そのログの直下呼出階層と呼ぶことにする。第 1 層の関数のみであり、そこからさらに下の階層で呼ばれる関数は含まない。したがって、直下呼出階層は、関数呼出の列となる。

前述の例において、ログ s のそれぞれの部分ログの直下呼出階層は以下のようになる。

- ログ s の直下呼出階層は [A, D]
- ログ s の関数呼出パス /A/B に対応する部分ログの直下呼出階層は [C]
- ログ s の関数呼出パス /D に対応する部分ログの直下呼出階層は [E, E]
- ログ s の関数呼出パス /D/E[1] に対応する部分ログの直下呼出階層は [] (空)

6.1.7 ログ s の関数呼出列階層表現

ログ中の関数呼出の列を階層表現で与える。前述のログ s は [A [B [C]], D [E, E]] と表現される。

6.1.8 構造化ログ

構造化ログとは、複数のログの集まりを包含する、制御構造を持った表現である。あるログの集まり S から制御構造を検出して得た構造化ログ L は、S を包含することは必要条件だが、S と等価であることは必要ではない (S に含まれないログが L に含まれることはありえる)。

以下のログ s1 と s2 からなるログの集まりを考える。

[ログ s1]

A

B

C

D

E

E

E

[ログ s2]

A

B

F

G

G

G

G

このログの集まりから構造化ログ x を得たと仮定する。

6.1.9 構造化ログにおける制御構造

制御構造は、loop, alt, opt の 3 種とする。

1. loop(ログ): ログに与えられた関数呼出列の複数回の繰り返し出現
   ログ x において、関数 D (/D) の中で関数 E が複数回繰り返し呼び出されている
2. alt(ログ 1, ログ 2): ログ 1 かログ 2 のいずれかの関数呼出列の出現
   ログ x において、関数 D の呼出 (/D) 以下の部分ログと、関数 F の呼出 (/F) 以下の部分ログのいずれかの関数呼出列が出現する
3. opt(ログ): ログに与えられた関数呼出列の出現もしくは全体の未出現
   ログ x において、関数 B (/A/B) の中で関数 C の呼出があるか、もしくは、ない

ログ s1 と s2 から得られる構造化ログ x は図 1 のようになる。

6.1.10 構造化ログ s の関数呼出列階層表現

構造化ログ x は [A [B [opt([C])]], alt(D [loop([E])], F [loop([G])])] と表現される。

6.1.11 関数呼出列 t1 と関数呼出列 t2 の距離

関数呼出列 t1 と関数呼出列 t2 の距離を、t1 を t2 に変形するのに必要な関数呼出の挿入・削除・置換の最小回数（つまり、編集距離、レーベンシュタイン距離）と定義する。関数呼出列の距

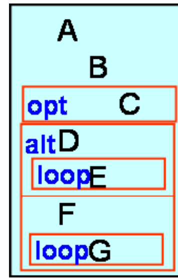


図 1　構造化ログ x

離は DP マッチングにより算出できる。

6.1.12　関数呼出列 t1 と関数呼出列 t2 がマッチする

「関数呼出列 t1 と関数呼出列 t2 がマッチする」とは、以下のいずれかと定義する（状況に応じて一つを選択する）。

- MS1：t1, t2 が完全一致する
- MS2：t1, t2 が包含関係にある (t1 から 0 個以上の呼出を除いた関数呼出列が t2 と MS1 マッチ)。よって、t1, t2 が MS1 マッチするならば必ず MS2 マッチする
- MS3：t1, t2 の距離が D 以下（D は定数とする、または t1, t2 のサイズに応じて決定する）

たとえば、t1=[A, B, B, C, D, D]、t2=[A, B, E, C, D, D] において、t1 と t2 の距離は 1 であり、M1 マッチではなく、M2 マッチではなく、M3 マッチである (D ≧ 1 のとき)。

6.1.13　ログ s1 とログ s2 の距離

ログ s1 とログ s2 の距離を、s1 を s2 に変形するのに必要なログの挿入・削除・置換の最小回数（つまり、編集距離、レーベンシュタイン距離）と定義する。ログの距離は DP マッチングにより算出できる。

6.1.14　ログ s1 とログ s2 がマッチする

「ログ s1 とログ s2 がマッチする」とは、以下のいずれかと定義する（状況に応じて一つを選択する）。

- M1：s1, s2 が完全一致する (直接呼出階層が MS1 マッチし、かつ、直接呼出階層の個々の関数呼出パスに対応するそれぞれの部分ログ同士が M1 マッチ)
- M2：s1, s2 の直接呼出階層が MS1 マッチする（子の相違は不問とする）
- M3：s1, s2 が包含関係にある (M2 または M4 であり、かつ、直接呼出階層の一致する関数呼出パスに対応するそれぞれの部分ログ同士が M3 マッチ)

- M4：s1, s2 の直接呼出階層が MS2 マッチする （子の相違は不問とする）
- M5：s1, s2 の距離が D 以下（D は定数とする、または s1, s2 のサイズに応じて決定する）

たとえば、s1=[A [B, B], C [D, D]]、s2=[A [B, E], C [D, D]] において、s1, s2 の距離 = 1 であり、M1 マッチではなく、M2 マッチであり、M3 マッチではなく、M4 マッチである、M5 マッチである (D ≧ 1 のとき)。

6.1.15　制御構造とログのマッチ

構造化ログが [F1, F2, ⋯, Fn] の形式で与えられているとする (Fi は構造化ログか、または制御構造)。制御構造は loop(L), alt(L1, L2), opt(L) のいずれかとする (L, L1, L2 は構造化ログ)。

「ログ s が構造化ログ x にマッチする」とは、以下のように定義する。

- x 中に制御構造がない場合 (x が単なるログ) → ログ s がログ x にマッチする
- x 中の Fi が制御構造 struct である場合:
  - struct = loop(L): s の先頭からの部分ログが [F1, F2, ⋯, Fi-1] にマッチし、かつ、それ以降の s の部分ログが L に 1 回以上マッチし、かつ、それ以降の s の末尾までの部分ログが [Fi+1, ⋯, Fn] とマッチする
  - struct = alt(L1, L2): s の先頭からの部分ログが [F1, F2, ⋯, Fi-1] にマッチし、かつ、それ以降の s の部分ログが L1 にマッチするか L2 にマッチし、かつ、それ以降の s の末尾までの部分ログが [Fi+1, ⋯, Fn] とマッチする
  - struct = opt(L): s の先頭からの部分ログが [F1, F2, ⋯, Fi-1] にマッチし、かつ、それ以降の s の末尾までの部分ログが L + [Fi+1, ⋯, Fn] とマッチするか [Fi+1, ⋯, Fn] とマッチする

たとえば、ログ s = [A [B [C]], D [E, E, E]]、構造化ログ x = [A [B [opt([C])]], alt(D [loop([E])], F [loop([G])])] のとき、s は x にマッチする。

6.1.16　構造化ログ x とログ s の距離

再帰的にすべての部分ログがマッチした時点で、x 中の個々の制御構造は特定のログに確定している。

- loop(L): L の何回の繰り返しがログ s とのマッチを成功させるかが確定
- alt(L1, L2): L1, L2 のどちらがログ s とのマッチを成功させるかが確定
- opt(L): L が出現するかしないか、どちらがログ s とのマッチを成功させるかが確定

すべてを確定したログ x' とログ s の距離が、構造化ログ x に対するログ s となる。

7 アルゴリズム

7.1 分岐構造の検出アルゴリズム (ログのリスト)

ログのリスト L から opt, alt フラグメントを検出するアルゴリズムを示す。

```
入力: ログのリスト L
出力: ログのリストから構造化ログへのマッピング m
while (L ≠ [])
  リスト S = []
  ログ w
  for ログ u in L
    リスト T = []
    for ログ v in L
      if (u == v) T に u を追加
      else if (u と v がマッチ) T に v を追加
      else break
    if (T ⊃ S) /* S より T の方が大きい */
      S = T
      w = u
  if (S == []) brek /* 何一つ相互にマッチするログがなかった */
  v = S の先頭要素
  T = S の残りの要素から w を削除したリスト
  構造化ログ sl
  for each w と v の一致しない箇所
      if  w 内のいずれかの関数を削除すれば v と一致する場合
        sl に opt(削除対象) を作成
      else if  w 内のいずれかの関数を何らかの関数に置換すれば v と一致する場合
        sl に alt(置換対象, 置換結果) を作成
  for ログ v in T
    for  each sl と v の一致しない箇所
      if  sl 内のいずれかの関数を削除すれば v と一致する場合
         sl に opt(削除対象) を作成
      else if  sl 内のいずれかの関数を何らかの関数に置換すれば v と一致する場合
         sl に alt(置換対象, 置換結果) を作成
  m = m + (S, sl) /* マッチするログのリストから作成した構造化ログへのマッピングを追
加 */
  L から S 中のログを削除
```

このアルゴリズムをフローチャートで図 2 に示す。

このアルゴリズムによる opt,alt の検出の例を図 3 に示す。

7.2 ログのリスト L から 構造化ログに変換するアルゴリズム

図 3 のアルゴリズムにログのリスト L を入力として与え、マッピング m を得る。

```
入力: ログのリスト L
出力: 構造化ログ r
opt,alt 検出のアルゴリズムにログのリスト L を入力として与え、マッピング m を得る
r = m (L)
```

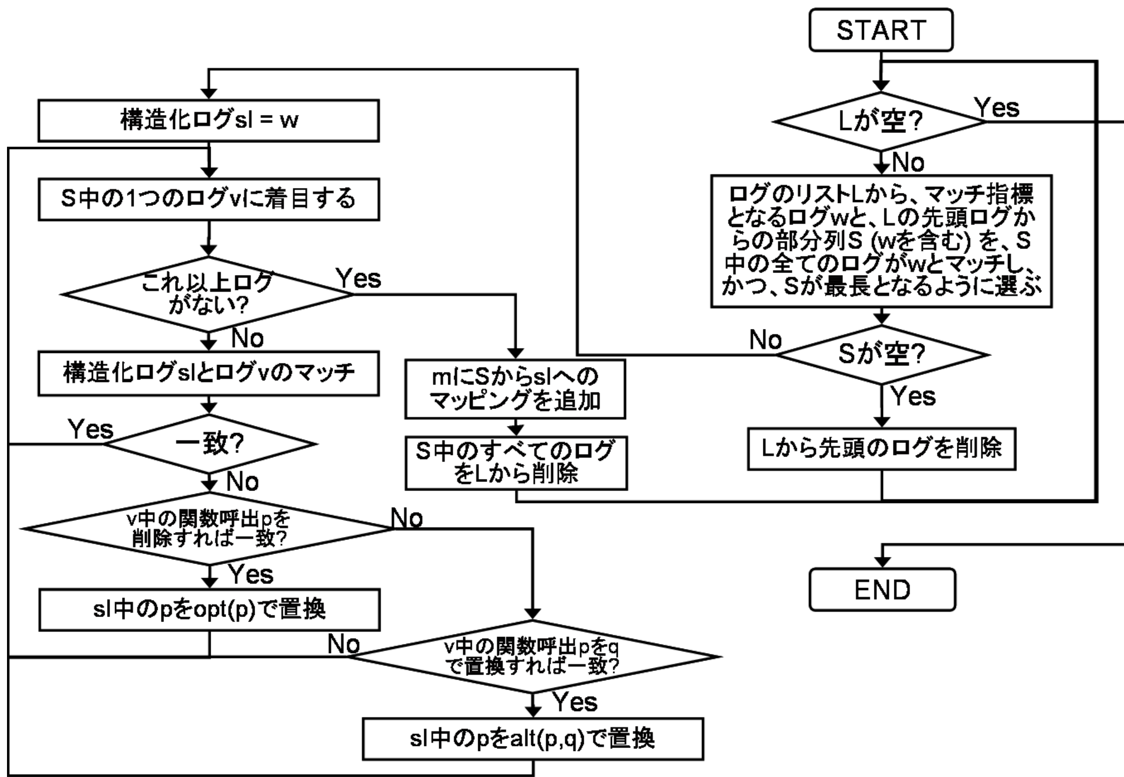


図 2　ログのリスト L から opt, alt を検出するフローチャート

7.3　ログ s から opt, alt フラグメントを検出するアルゴリズム

```
入力: ログ s
出力: 構造化ログ r
r = s のコピー
連想配列 h : 関数 ? ログのリスト
d = r の最大深度
for  x = 1 .. d
  c = 第 x 層のログの親の個数
  for  w = 1 .. c
    f = 第 x 層のうち w 番目の親
    t = f の子であるログ
    h [ f ] に t を追加する
for  f  in  h のキー
  上記のアルゴリズムにログのリスト h [ f ] を入力として与え構造化ログ u を得る
  if (u が空ではない)
    r 中に出現する f の呼出以下のログを u で置き換える
```

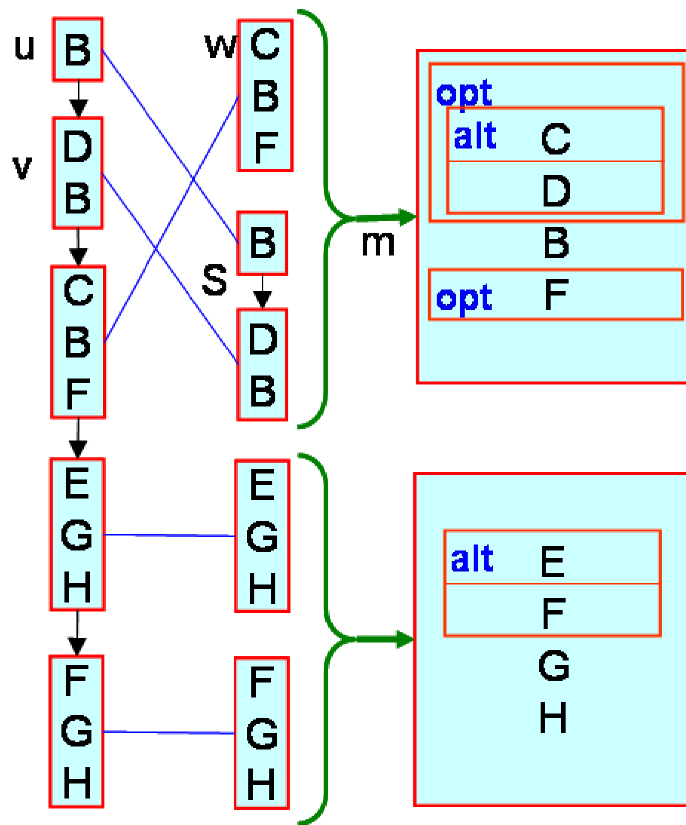


図 3　ログのリスト L からの opt, alt の検出例

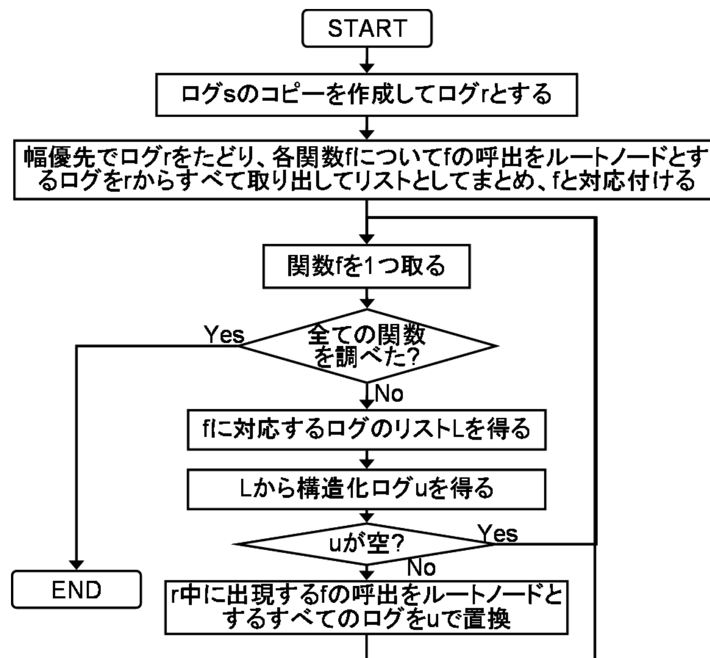


図 4　ログ s から opt, alt を検出するフローチャート

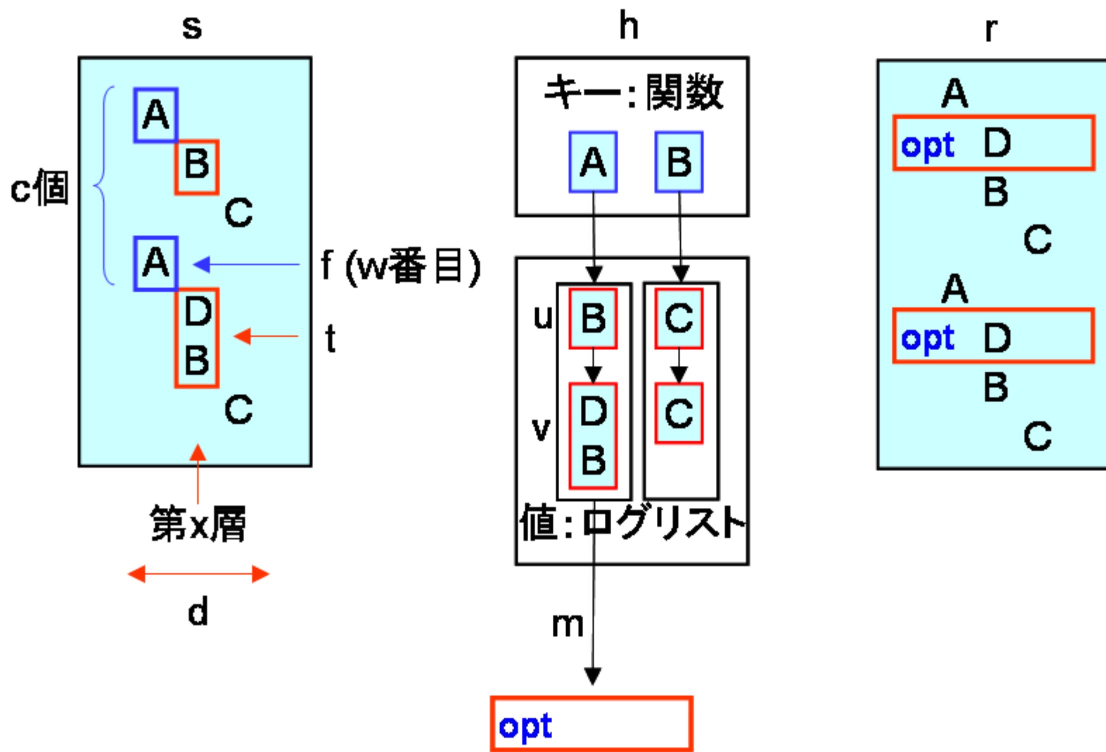


図 5　ログ s からの opt, alt の検出例

このアルゴリズムをフローチャートで図 4 に示す。

このアルゴリズムによる opt,alt の検出の例を図 5 に示す。

7.4　ループ構造の検出アルゴリズム (直下呼出階層)

ログ s の直下呼出階層から loop フラグメントを検出するアルゴリズムを示す。図 5 のアルゴリズムを内部で用いる。

```
入力: ログ s,
      loop の検出に必要な最低繰り返し回数 m (>1),
      検出の打ち切り閾値となる編集距離 o (>=0)
出力: 構造化ログ r
r = s のコピー
t = r の直下呼出階層
n = t の長さ
z = 1
while ( z < n )
  z++
  start = z-(n-z+1)/(m-1)-o
  if (start < 1) start = 1
  for  y = start .. (z-1)
    a = z
    ログ u = t 内の y 番目から z-1 番目までのログ
    ログのリスト L = [u]
    構造化ログ p = u
    while ( a < n )
      d = o /* 0 ではなく o(最大許容編集距離) */
      ログ x = [] (空)
      h = p の直下呼出階層の長さ
      flag = false
      for  k = h-o .. h+o
        b = a+k-1
        ログ v = t 内の a 番目から b 番目までのログ
        if ( v が p にマッチする)
          dn = v と p の編集距離
          if ( x が空 または d>dn)
            x = v
            d = dn
          ログのリスト Ln = L + [v]
          ログ s から opt, alt を検出するアルゴリズムに Ln を与え、構造化ログ q を得る
          if ( t 内の z+k 番目から、q とマッチする部分ログがある)
            flag = true
            L = Ln
            p = q
            x = v
            break
      if ( ! flag)
        L に x を追加
        p = ログ s から opt, alt を検出するアルゴリズムに L を与えて得た構造化ログ
      if ( x が空ではない)
        a += x の直下呼出階層の長さ
      else
        break
    if ( L 中のリストの個数 >= m)
      r 中の L を loop(p) で置き換え
      z += L 中のログの直下呼出階層の長さの総和
      break
```

このアルゴリズムをフローチャートで図 6 に示す。

このアルゴリズムによる loop の検出の例を図 7 に示す。

7.5 ループ構造の検出アルゴリズム

ログ s から loop フラグメントを検出するアルゴリズムを示す。図 7 のアルゴリズムを内部で用いる。

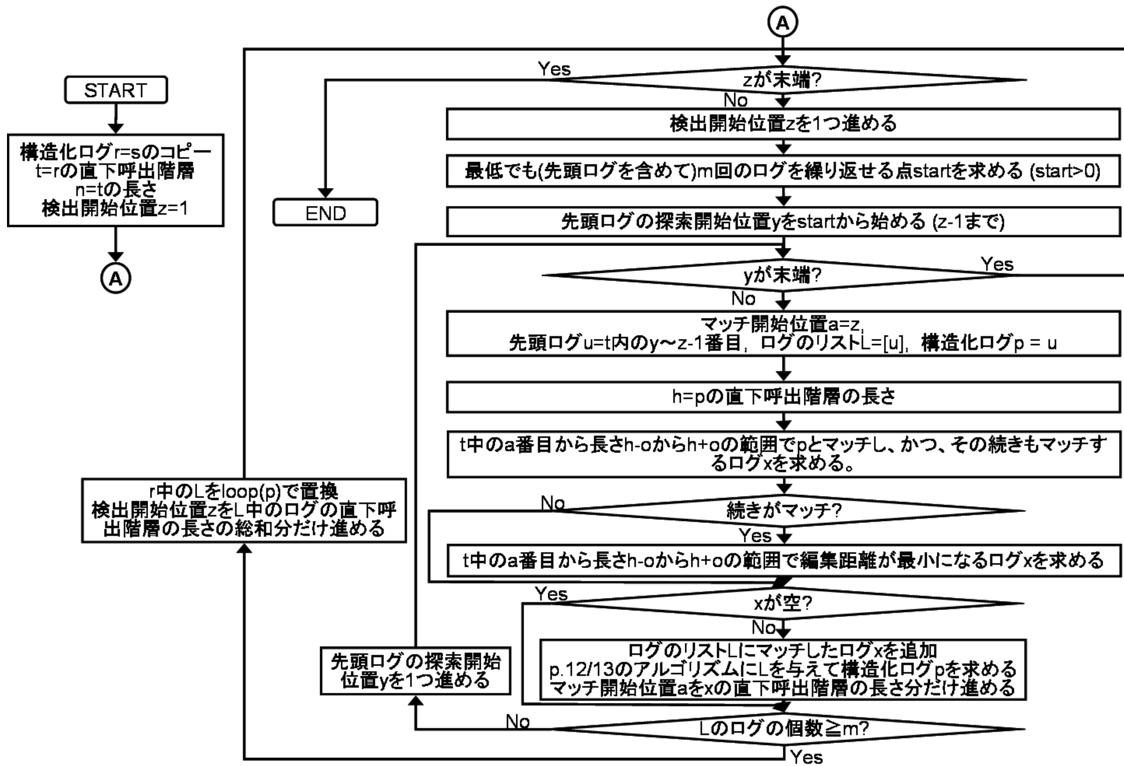


図 6　ログ s の直下呼出階層から loop を検出するフローチャート

入力: **ログ** s,
　　loop **の検出に必要な最低繰り返し回数** m (>1),
　　検出の打ち切り閾値となる編集距離 o (>=0)
出力: **構造化ログ** r
r = s **のコピー**
r **を幅優先で探索（関数呼出** f)
　f **を第** 0 **層とする** r **の部分ログ** p
　ログ s **の直下呼出階層から** loop **を検出するアルゴリズムの入力に** p **を与えて構造化ログ** q **を得る**
　r **中の** p **を** q **で置き換える**

このアルゴリズムをフローチャートで図 8 に示す。

このアルゴリズムによる loop の検出の例を図 9 に示す。

参考文献

[1] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.

[2] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions.

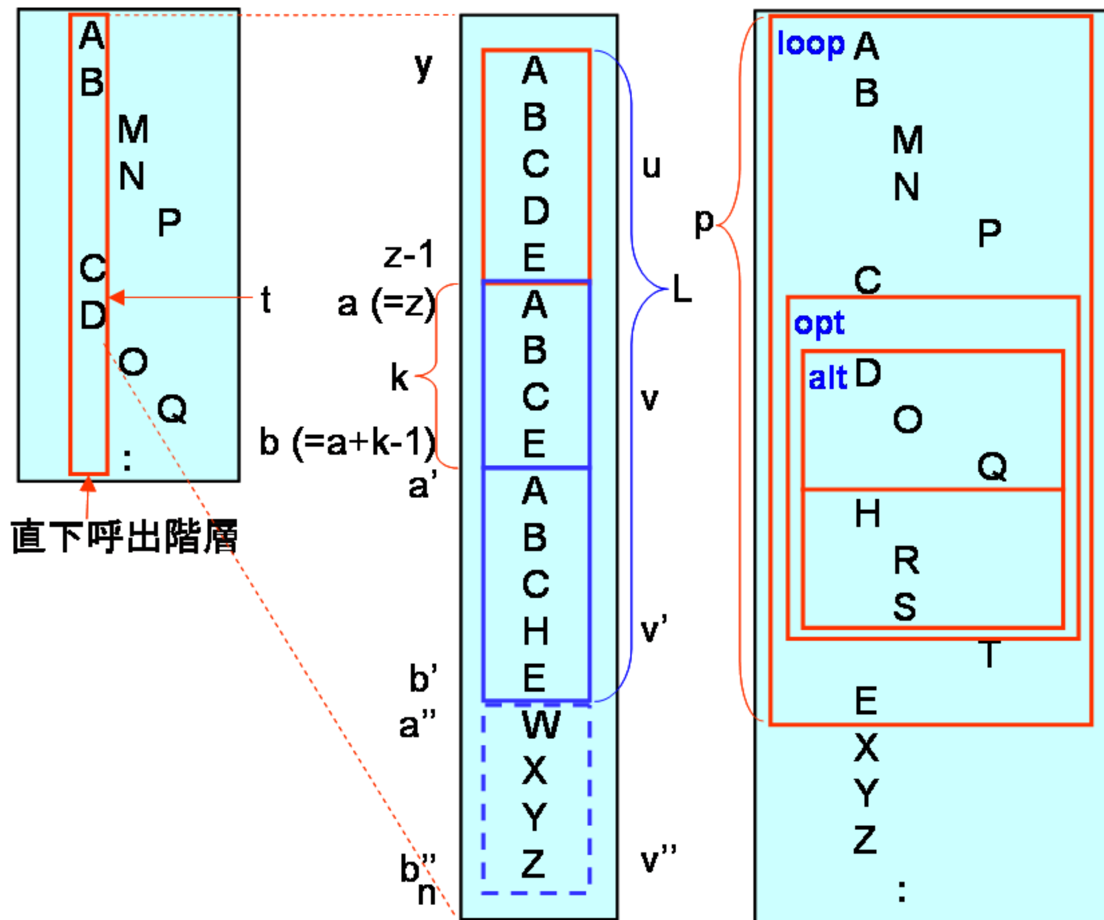


図 7　ログ s の直下呼出階層からの loop の検出例

In *Proceedings of the 19th International Conference on Software Engineering (ICSE 1997)*, pages 360–370, Boston, Massachusetts, USA, May 1997. IEEE Computer Society / ACM.

[3] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In S. Diehl, editor, *Software Visualization (LNCS 2269)*, pages 647–650. Springer-Verlag, 2002.

[4] 中村宏明, 小野康一, and 石川浩. *Method for Generating Sequence Diagrams from Source Code*, 2008. JP9-2008-0082.

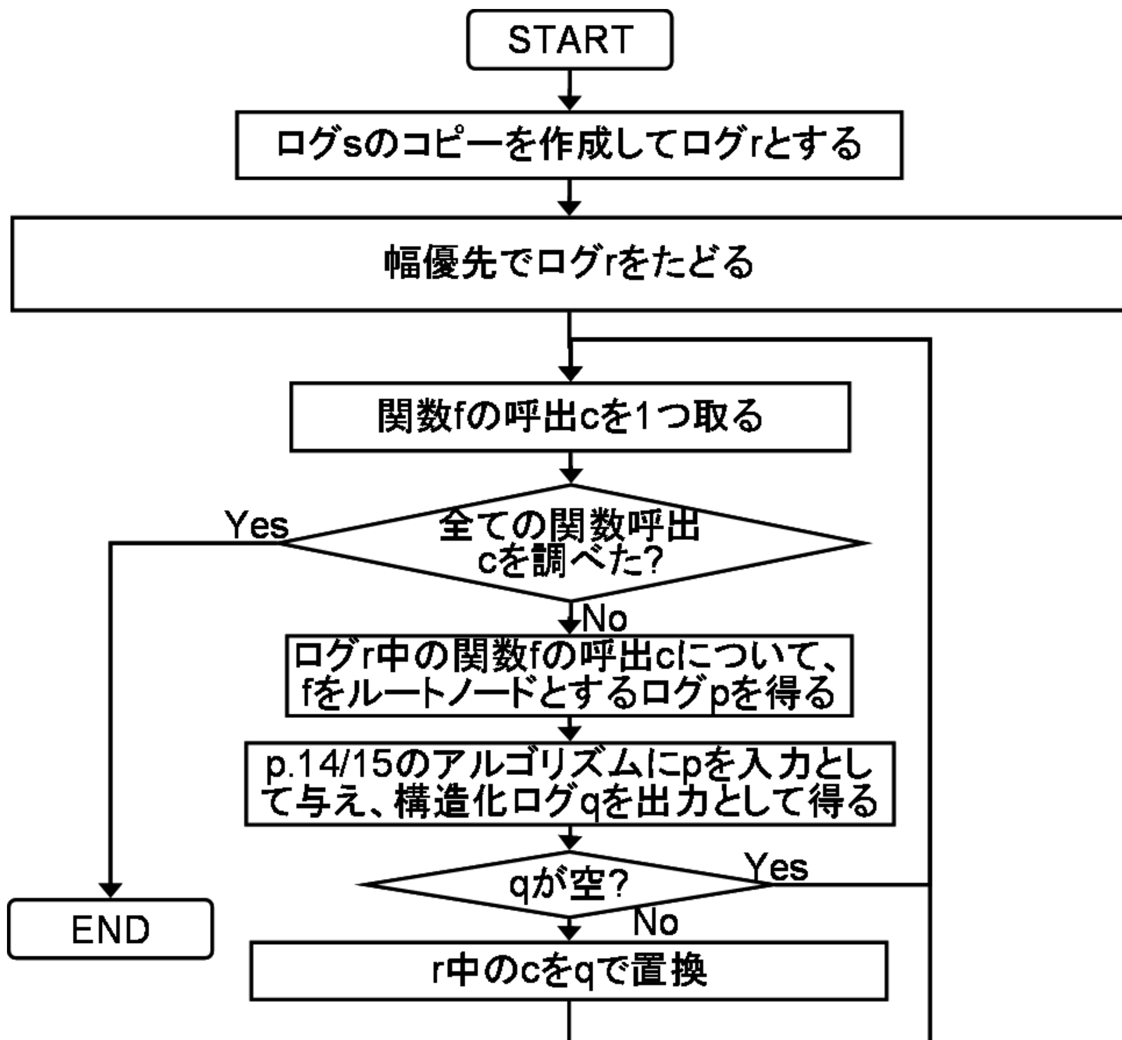


図 8　ログ s の直下呼出階層から loop を検出するフローチャート

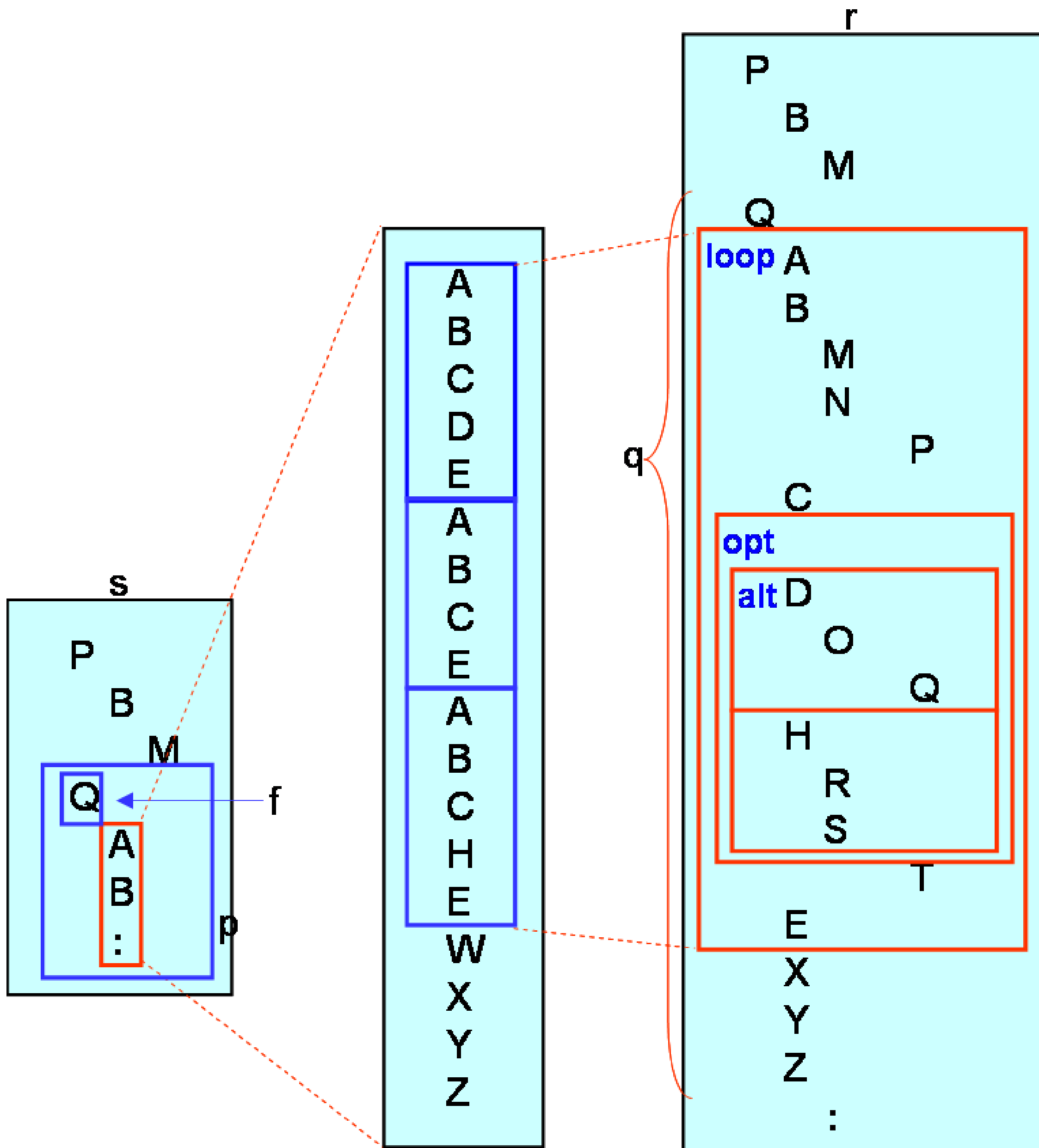


図 9　ログ s の直下呼出階層からの loop の検出例