Vol.2012-SE-175 No.16
2012/3/16

プログラマに思い込みを気付かせるデバッグ支援システムの実現に関する一考察

後藤邦仁[†]　　太田剛[††]

デバッグ作業が困難を伴う理由は，技術的な問題よりもむしろ，勘違いや思い込みをしている事実に，プログラマが気付く機会がなかなか訪れない点にある，と著者は考えている．本稿では，思い込みをしている事実をプログラマに気付かせる支援システムの実現方式を述べる．この方式では，プログラマが思い込みを起こしている箇所を，組み合わせ最適化問題に帰着させて解く．この方式に従って小規模な机上実験を行った結果に基づいて，計算量の問題を含めて実現可能性に関して議論する．本稿の方式が既存の研究と異なる特徴は，プログラムの実際の動作をプログラマが評価する際に，意図通りに動作しているか否かの判定を，時に誤ってもよいという点にある．

A feasibility study on a debugging support system that can remind a programmer of misunderstanding on his/her program

KUNIHITO GOTO[†]　TSUYOSHI OHTA[††]

The authors believe that difficulty of debugging comes from the fact that programmers have few chances to recognize his/her misunderstanding or wrong assumption on his/her program rather than technical issues. We describe an implementation plan to build a support system which can remind a programmer of his/her misunderstanding or wrong assumption. In this plan, we map the subject to a combinatorial optimization problem and solve it. We also describe the result of a small desk experiment and discuss about feasibility including computational complexity. Our main contribution is to show the implementation plan that allows programmers to make mistake and to misunderstand on his/her program's behavior.

1. はじめに

デバッグ作業は，ソフトウェア開発の一連の工程の中でもとりわけ時間がかかり，人的コストの大きい部分だと言われており，テスト工程と合わせて全体の 45%を占めるとの報告もある[1]．また，その作業は属人性が強く，経験の差が効率の差に現れてくることもよく知られている．

そのため，デバッグ作業を支援するための環境やツールがいろいろと提案され，実際に使われてきた．現在広く使われているツールがシンボリックデバッガであり，プログラムの実行を制御する手段と，実行途中での変数値を調べたり変更したりする手段をプログラマに提供している．統合開発環境と呼ばれている環境には，ほぼこのツールが組み込まれている．シンボリックデバッガは確かに有用ではあるが，プログラムをどこで停止させどの変数値を調査するかを決めることは，完全にプログラマに任されており，知的なガイドを与えるわけではないという意味で原始的なツールの域を出てはいない．また，シンボリックデバッガ使用中によくある，進めすぎてしまった実行を元に戻したい，この変数にこの値を代入したところまで戻したい，といったプログラマの要求に応えるための逆実行デバッガ（可逆デバッガ）を実現する研究もある．

一方で，属人的なデバッグ工程を，科学的手法によって系統的に，そして可能な限り自動化して支援しようとする研究が行われている．その嚆矢は，Shapiro による，論理型言語を対象とした algorithmic program debugging[2]であろう．これを手続き型言語に拡張した研究[3]もある．以来，誤り現象が関係する範囲だけを分離（isolation）し，原因を特定するところまでを系統的に，また一部自動的に実施するための研究が行われてきた．例えば，プログラムスライシング[4],[5],[6], delta debugging[7], Why line[8], DAIKON[9], DIDUCE[10], モデルチェッカの適用[11]などである．このあたりは Zeller の著書[12]に詳しい．

とは言うものの，これまでに実現されたシステムは，理論的には確かにその通りであっても，現実には使いにくい，もしくは使えないというのが正直な評価であろう．その一番の理由として我々は，これまでの研究がプログラマの能力に対して過剰な要請（期待）をしていることにあると考える．具体的には，プログラマに対して，表示された結果や実行途中の計算状態が正しいものなのかそうでないのかを，その場ですぐに yes/no で評価できること，またその評価は将来にわたって常に正しく，訂正されることはないこと，などが要請されているのである．

[†] 静岡大学大学院情報学研究科
Graduate School of Informatics, Shizuoka University
[††] 静岡大学情報学部
Faulty of Informatics, Shizuoka University

ところが現実には，プログラマは勘違いをすることもあれば，思い込みによって誤ることもある.「こともある」どころか,デバッグ工程においてはそれが頻繁に起きる．そしてその勘違いや思い込みに気が付かないままデバッグ作業を行うため，本来は確認を必要とする部分であるのに正しいと信じて調査対象からはずし，袋小路にはまりこんでバグの分離や特定の作業がさっぱり進まない結果となるのである．勘違いや思い込みがあったことに気付いたことによって，それまで遅々として進まなかったデバッグ作業が劇的に進んだという経験は，プログラミング経験のある者であれば誰もが持っているであろう．企業内のペアプログラミングでバグの作りこみが 15%以上削減できたとの報告[13]は，ペアの片割れの言葉がきっかけとなって思い込みに気付いた結果だと解釈することもできる．

そこで我々は，プログラマに「自分はここで勘違いや思い込みをしていたのだ」と気付かせるような仕掛けを用意することで,デバッグを支援するシステムを構想した．本稿ではその実現方式の概略を示し，事例に基づいて実現可能性について議論する．

2. 実現方式の概略

2.1 プログラマに対する要請

我々が構想しているシステムがプログラマに対して要請することは次の点である．この要請はかなり緩いものであり，プログラマに完璧な評価を要求しないという点において，既存の研究における要請とは大きく異なる．

- プログラマにとって，プログラムが自分の意図通りに動作しているか否かを評価できるような実行時点がある．そのような実行時点は，システムがプログラマに提示するのではなく，プログラマの自由意思で選択できる．言い換えれば，プログラマにとってわかっていることだけを回答すればよい．
- 思い込みや勘違いのため，上記評価が時に誤ることを許す．

2.2 プログラマの意図のシステム内への取り込み

デバッグ工程におけるプログラマの思い込みは，2 つの理由から生じている．ひとつはプログラム実行の様子を観察している際に，自らの意図していたものとは異なる動作をしているにもかかわらず，これを意図通りに動いていると思い込むこと，もしくは意図通りに動作しているにもかかわらず，これを意図通りに動いていないと思い込むことであり，もうひとつは，自分の書いたプログラムは自分の意図していることを正確に表現していると思い込むことである．

これらふたつを同じ枠組みの中で扱うために，グラフを構成して利用する．その概要を図 1 に示し，利用法を以下に述べる．

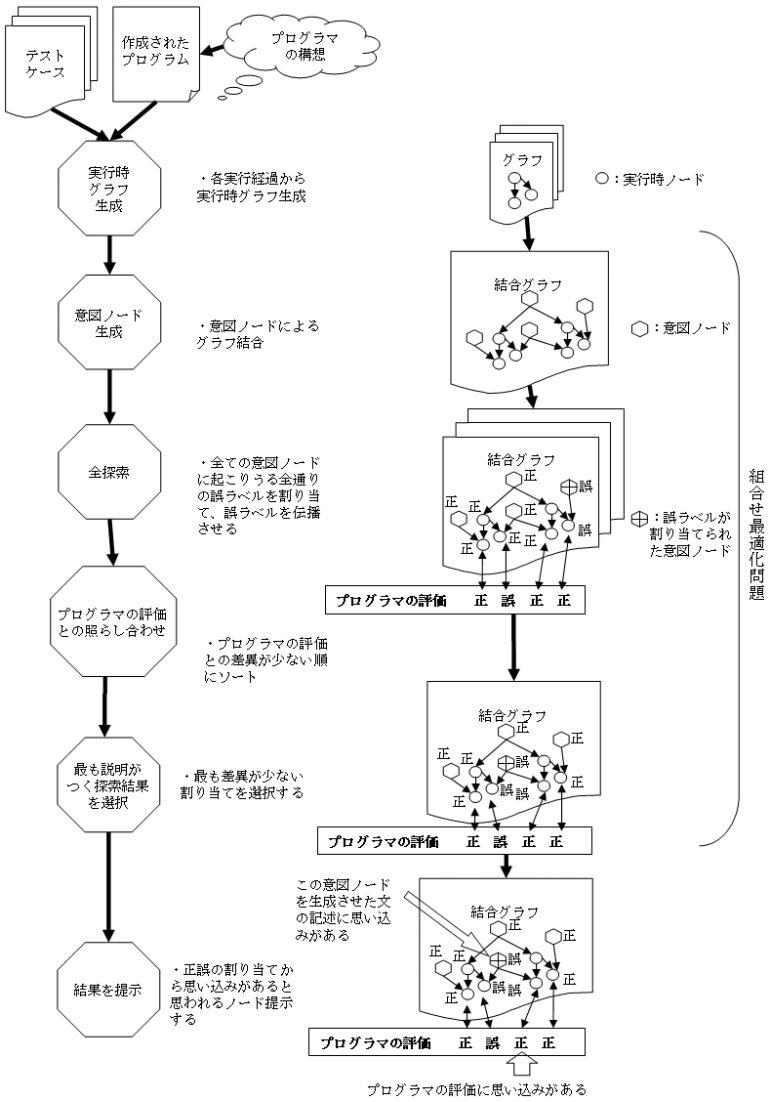

図 1 処理手順とグラフ利用法

プログラム実行時のプログラマの思い込みを扱うため，プログラムの実行過程を有向グラフに表現する．この有向グラフのノードを**実行時ノード**と呼び，プログラム文の実行インスタンスを表す．エッジはそれら実行インスタンスにおけるデータおよび制御の依存関係を表す．すなわち，実行時ノード A で値が定義された変数が実行時ノード B で参照されたなら，A から B へデータ依存エッジが引かれる．また，実行時ノード C の条件判定が実行時ノード D の実行を左右したなら，C から D へ制御依存エッジが引かれる．こうして作成された有向グラフを**実行時グラフ**と呼ぶ．このシステムにおいては，テストケース 1 つに対して 1 つの実行時グラフを生成するので，複数のテストケースを扱う場合には，実行時グラフが複数存在することになる．

プログラム記述時のプログラマの意図をシステム内で扱うため，プログラムの各文に対して 1 つのノードを生成する．このノードを**意図ノード**と呼ぶが，そこにどのような意図があったのかについて，システムは感知しない．その文が記述されているという事実から，そこにはプログラマの何らかの意図が込められているはずだということを表しているにすぎない．プログラム上の同一文から生成されたノードが実行時ノードの中に複数あるが，それらの全てに対して，その文に対する意図ノードからエッジが引かれる．意図ノードによって複数の実行時グラフが結合され，ひとつの大きなグラフとなる．この大きなグラフを**結合グラフ**と呼ぶ．

2.3 組み合わせ最適化問題への変換

こうして得られた結合グラフの各ノードに対して，正／誤のラベルを割り当てる問題を考える．ここで，正ラベルはそのノードがプログラマの意図通りに動作することを，誤ラベルは意図通りでないことを表すものとする．プログラムの誤りが実行時に次々と伝播していくのをモデル化するため，誤ラベルが割り当てられたノードからは，エッジを順方向にたどることによって誤ラベルが伝播していくものとする．

さて，2.1 節の要請に基づいて，プログラマは自分がわかる範囲で，実行状態の正誤を評価[a]し，いくつかの実行時ノードに対して正／誤ラベルを貼り付けることができる．一方，それとは独立に，システムが意図ノードに正／誤ラベルを機械的かつ系統的に割り当てて[b]誤ラベルを伝播させることもできる．後者におけるラベルの割り当て方は，単純に考えると 2 の意図ノード数乗だけ存在することになるが，その中には，プログラマが評価したラベルとの食い違いが少ないものから多いものまで，様々なものが現れてくる．このとき，食い違いが最も少なくなる割り当て方が存在する．そのような割り当て方を見つけることは，依存関係の制約のもとで，食い違ったノード数を最小にする組み合わせ最適化問題を解くことに他ならない．

2.4 最適割り当ての解釈

こうして得られた最適なラベル割り当てが何を意味するか考えてみよう．

システムはあらゆる可能性の中から，プログラマが現在主張していること[c]を最も上手く説明できるラベル割り当てを見つけ出してきている．したがって，この最適な割り当て結果に対して次のような解釈が可能である．

A　プログラマの評価との間で食い違いがなかった場合

意図ノードへのラベル割り当てが，すべてのテストケースにおいてプログラマの主張するプログラム実行状況を完全に説明できたことを意味する．つまり，誤ラベルが割り当てられた意図ノードに何らかの問題があるはずだということである．これはすなわち，その意図ノードが表しているプログラム文に対して，プログラマの思い込み（すなわち欠陥）が含まれている可能性が極めて高いことを意味している．

B　プログラマの評価との間で食い違いがあった場合

食い違いのあった実行ノードにおいて，意図通りに動作しているか否かの評価をプログラマが誤った可能性がある．そのことを理解したシステムは，プログラマに対して，その実行ノードにおける評価の再確認を促すことができる．これによって，「思い込みをしているかもしれないという事実に気が付く機会」をプログラマに提供できる．このとき，確かに思い込みであったとプログラマが確認すれば，A の場合に還元されて欠陥の位置が判明する可能性がある．再確認を促してもやはりまちがっていないとプログラマが回答したときは，そのラベル割り当てを捨てて，次善の最適ラベル割り当てを選んで同様のことを行うこともできる．

3. 事例実験

3.1 本実験で扱う範囲

2 節で示した方式の実現可能性を調べるために，机上での事例実験を行った．プログラマは出力文に対してのみ評価を行い，プログラマの評価間違いの数は 0 個～1 個に限定した．また，誤ラベルの伝播の仕方は，データ依存と制御依存の区別をせず，同様に伝播していく事とした．最終的にプログラマの 2 種類の思い込みを指摘し，プログラマの思い込みに対して気づきを与えることができたかを評価する．

3.2 実験対象プログラムについて

今回の事例実験で用いた実験対象プログラムを図 2 に示す．実験対象プログラム は

a) プログラマが自ら判断し，実行時ノードに正/誤のラベルを付けることを「評価する」ということにする．
b) システムが機械的かつ系統的に意図ノードや実行時ノードに正/誤ラベルを付けることを「割り当てる」と言うことにする．
c) 2.1 節の要請により，この主張には誤りが含まれていてもよいことに注意する．

C 言語で記述されており，3 つの正の整数値を入力して最大値と最小値を出力するプログラムである．このプログラムには 13 行目の代入文に欠陥が含まれている．`z = a[1]` ではなく `z = a[0]`こそがプログラマが想定した理想の動作をする．`y` は仮の最大値を保持する変数であり，`z` は仮の最小値を保持する変数である．そして，最大値の計算を行う条件文と最小値の計算を行う条件文が `else` によって一つの条件文にまとめられているので（17 行目），ループ 1 回につき最大値の計算か最小値の計算

```
1 行目   #include <stdio.h>
2 行目
3 行目   int main(void){
4 行目     int x,y,z,i;
5 行目     int a[3];
6 行目
7 行目     for(i=0;i<3;i++){
8 行目       scanf("%d",&x);
9 行目       a[i] = x;
10 行目  }
11 行目
12 行目  y = 0;
13 行目  z = a[1]; //追加した欠陥
14 行目  for(i=0;i<3;i++){
15 行目    if(y<=a[i]){
16 行目      y = a[i];
17 行目    }else if(z>=a[i]){
18 行目      z = a[i];
19 行目    }
20 行目  }
21 行目
22 行目  printf("MAX = %d¥n",y);
23 行目  printf("MIN = %d¥n",z);
24 行目
25 行目  return 0;
26 行目}
```

図 2　実験対象プログラム

のどちらかしか行うことができない. そのため, 14 行目からのループ内では常に `z<=y` が成立していなければ正しい計算ができない. ところが, 13 行目で仮の最小値を `a[1]` としているので，入力の際に `a[0]`に真の最小値が与えられていた場合に，次のような問題が発生して故障が明らかになる．つまり，1 回目のループでは，15 行目の条件式が必ず真になるので（`y=0` かつ `a[0]`は正の値である），18 行目が実行されることはない．つまり，真の最小値が仮の最小値として扱われることが決して起こらない．

3.3 実行時グラフおよび結合グラフを生成する

表 1 は，実験対象プログラムに様々な入力を与えたときの実際の出力と，プログラマが想定している出力を表に表したものである. テストケースは, 分岐網羅率が 100%になるように 6 つ作成した．そして，テストケースごとの実行経過を記録し，その実行経過から実行時グラフを生成する．その後 2.2 節で示したように意図ノードを生成した．このとき生成された意図ノードの数は 18 個であった.

表 1　実験対象プログラムの実行結果

テストケース	入力	出力された値		プログラマが意図した値		実行時グラフのノード数
		Max	Min	Max	Min	
Test1	1,2,3	3	2	3	1	34
Test2	1,3,2	3	2	3	1	35
Test3	2,1,3	3	1	3	1	35
Test4	2,3,1	3	1	3	1	35
Test5	3,1,2	3	1	3	1	35
Test6	3,2,1	3	1	3	1	36

3.4 組合せ最適化問題を解く

意図ノードに対し，起こりうる誤ラベルの割り当て方を全通り行う．今回生成された結合グラフには意図ノードが 18 個あり，与えられるラベルは「正」と「誤」の 2 種類なので，ラベルの割り当て方は 2 の 18 乗の 262144 通りである．その後，2.1 節で示したように, プログラマがわかる範囲で出力文（22, 23 行目にあたる実行時ノード）に対して正/誤の評価を行う．そして，システムが割り当てたラベルから伝播した誤ラベルと，プログラマが行った評価との食い違いが最も少なくなるような意図ノードへのラベルの割り当て方を机上で調査した.

表 2　プログラマの評価と欠陥の位置とラベルの食い違いの対応表

プログラマが評価を間違えたノード		欠陥を指摘できたか	ラベルの食い違い	
テストケース	実行時ノード		実行時ノードの行番号	思い込みを指摘できたか
なし		○ 13 行目	Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	×
Test1	22 行目	○ 13 行目	Test1 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	× 17 行目	Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	×
Test2	22 行目	○ 13 行目	Test2 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	○ 13 行目	Test2 の 23 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
Test3	22 行目	○ 13 行目	Test3 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	○ 13 行目	Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	×
Test4	22 行目	○ 13 行目	Test4 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	○ 13 行目	Test3 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	×
Test5	22 行目	○ 13 行目	Test5 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	○ 13 行目	Test3 の 23 行目,Test4 の 23 行目,Test6 の 23 行目	×
Test6	22 行目	○ 13 行目	Test6 の 22 行目,Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目,Test6 の 23 行目	○
	23 行目	○ 13 行目	Test3 の 23 行目,Test4 の 23 行目,Test5 の 23 行目	×

※下線にはプログラマに思い込みを気づかせることが可能なノードの行番号を示す.

表 3　表 2 の 2 行目に対する最適解の全て

誤ラベルがついた意図ノードの数	誤ラベルがついた意図ノード
1	13 行目
	23 行目
2	13 行目，17 行目
	13 行目，18 行目
	13 行目，23 行目
	17 行目，23 行目
	18 行目，23 行目
3	13 行目，17 行目，18 行目
	13 行目，17 行目，23 行目
	13 行目，18 行目，23 行目
	17 行目，18 行目，23 行目
4	13 行目，17 行目，18 行目，23 行目

3.5 結果を解釈する

プログラマが評価を間違えた実行時ノードとその時に指摘される欠陥の位置と指摘されるラベルの食い違いが生じた実行時ノードとの対応表を表 2 に示す. この表は,「プログラマが評価を間違えたノード」の欄で，そのテストケースとプログラマが評価間違いをしたと仮定した実行時ノードを示している.「欠陥を指摘できたか」の欄には, 本手法によって, 行番号の意図ノードに欠陥があると指摘できたかを示しており，その行番号が妥当であるならば「○」，妥当でないなら「×」と記した.「ラベルの食い違い」の「ノードの行番号」欄では，本手法によって，プログラマの評価と自動的に割り当てられたラベルの伝播結果に食い違いが生じたノードの行番号を示し，その右の「思い込みを指摘できたか」の欄では，その食い違いの中にプログラマが評価を間違えたものが含まれていれば「○」を，ない場合は「×」と記した.

表 2 はプログラマが 0 個〜1 個の評価間違いをしたときの組合せ最適化問題のそれぞれの最適解を表している.「欠陥を指摘できたか」の欄が示す行番号は，プログラマの評価を最も上手く説明できるように意図ノードに誤ラベルを割り当てたところなの

で，その行番号の文に対して，プログラマがなんらかの思い込みを持って記述している可能性があることを意味している．また，「ラベルの食い違い」の「実行時ノードの行番号」の欄が示す行番号は，システムのラベル割り当てと，プログラマの評価とが異なったところなので，プログラマがなんらかの思い込みを持ってこの実行時ノードを評価した可能性があることを意味している．つまり，「実行時ノードへの行番号」から生成される実行ノードへのプログラマの評価は，間違っている可能性があると解釈できる．まとめると，組合せ最適化問題の最適解はプログラマのなんらかの思い込みの影響を受けている可能性のある箇所を示している．この中に「プログラマが評価を間違えたノード」の行番号が含まれていれば，プログラマに思い込みを指摘できることになる．

ここでプログラマが Test2 の 23 行目の評価を間違えた場合（表 2 の 5 行目）について説明する．そのときの意図ノードへの最適な誤ラベルの割り当て方は，表 3 に示す 12 通りがある．表 3 の最左欄は，意図ノードに誤ラベルが割り当てられた数を示し，その右の欄は，誤ラベルが割り当てられた意図ノードにあたる行番号を示している．そのうち，今回はプログラムの 13 行目の意図ノードに誤りラベルが割り当てられた場合（表 3 の 1 行目）について，その時の結合グラフの一部分を図 3 に示して詳しく説明する．図 3 では実行時ノードを楕円で，意図ノードを六角形で表している．エッジの近くに書かれている文字は，参照された変数とその値を表している．プログラマは色付けられている printf 文の実行時ノードに対して正誤の評価を行っている(それぞれの評価は図 3 内に記した)．

この時の組合せ最適化問題を解くと，13 行目の意図ノードに誤ラベルが割り当てられる（表 2 の 5 行目）．つまり，13 行目にプログラマの意図が正しく反映されなかったことを意味していると解釈すれば，プログラマが現在置かれている状態が最も上手く説明できる．つまり，これは 2.4 節で述べた通り，プログラマが意図していることが正確に表現できていると思い込んでいる状態であることを示している．プログラマにこの思い込みを指摘する例としては，「13 行目に欠陥がある可能性があります」などが考えられる．

また，図 3 で示されているノードには全て 13 行目の意図ノードから誤ラベルが伝播しており，プログラマが「正しい」と評価したノード（薄い色で塗りつぶされたノード）との間でラベルの値が食い違っている．この例では Test2 の 23 行目の評価を間違えたことにしていたので，プログラマが自らの意図していたものとは異なる動作をしているにもかかわらず，意図通りに動いていると思い込んだかもしれないことを検出できたと言える．これをプログラマに指摘する例としては，「あなたは Test2 の 23 行目と Test3 の 23 行目と Test4 の 23 行目と Test5 の 23 行目と Test6 の 23 行目を正しいと評価しましたが，その評価は本当に正しいですか？」などが考えられる．

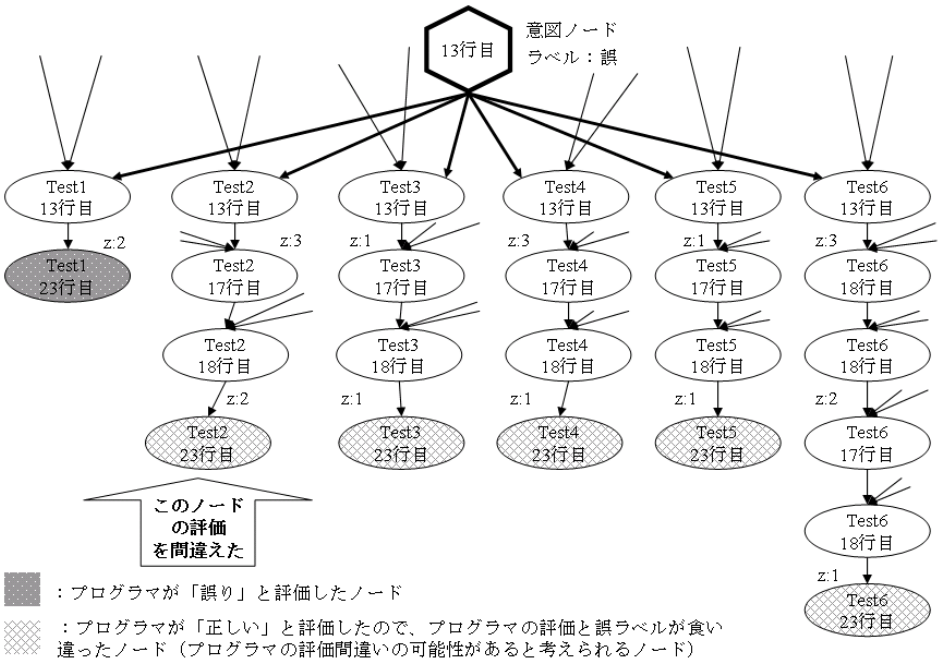

図 3　Test2 の 23 行目の評価をプログラマが間違えた場合の結合グラフの一部

4. 議論

4.1 事例実験の妥当性

今回の事例実験の結果の妥当性を表 2 の「欠陥を指摘できたか」の欄（表 2 の右から 3 列目）と「思い込みを指摘できたか」の欄（表 2 の最右欄）に示した．本事例実験の妥当性とは，プログラマの持つ思い込みをプログラマ自身に気づかせることが可能な指摘ができるかどうかである．プログラマが評価したラベルとシステムの割り当てたラベルとの間で食い違いが生じたノードには，プログラマのなんらかの思い込みが含まれている可能性がある指摘は，13 個中 7 個（表 2 の最右欄の丸の数）上手く指摘できた．しかし，たとえ確かに思い込みであったとプログラマが確認しても，プログラマが割り当てたラベルとの間で食い違いがなくなる状態に還元される場合は見つ

けられなかった．これは，実際には故障が必ずしも全てのノードに伝播するわけではないことが原因だと考えられる．つまり，本手法では，欠陥から発生した故障は必ず伝播すると仮定しているので，故障が顕在化するノードと顕在化しないノードが両方ともに現れると，確かに思い込みであったとプログラマが確認しても，プログラマが割り当てたラベルとの間で食い違いがなくならない場合があると考えられる．だが，このような情報をプログラマに指摘することによって,「思い込みをしているかもしれないという事実に気が付く機会」をプログラマに提供することは可能であったといえる．

また，表 2 の 3 行目の「欠陥を指摘できたか」については×がつけられているが，欠陥を作りこんだ 13 行目ではなく，17 行目の意図ノードに誤ラベルを割り当てるのが数理的意味において，最適であったからである．この結果を受けて，ソースコード上の 17 行目を調べてみた結果，13 行目を修正しなくても 17 行目を修正することによって，当初実現しようと試みていた機能を正しく実装することが可能であることがわかった．具体的には，`}else if(z>=a[i]){`の `else` を削除し，`} if(z>=a[i]){` と修正する．これで，1 回のループの中で，最小値の計算と最大値の計算の処理を両方行えるようにすることで，`z<=y` が成立していなくても問題が発生せず，欠陥を修復できる．このことはつまり，本稿で述べられてきた考え方が著者の持っていた思い込みに気づく機会を与えてくれたと言っていい．よって，今回の事例実験においては，17 行目にプログラマの思い込みがあると指摘することは，プログラマの持つ思い込みをプログラマ自身に気づかせる機会を提供できていると考えられる．

4.2 プログラマに指摘する情報量について

今回の事例では，プログラマの評価間違いを最大 1 つに限定した．この条件のもとであっても表 2 に示したように，プログラマが評価したラベルとシステムが割り当てたラベルとの間で食い違いが生じた実行時ノードの数は，3 個〜5 個となった．本手法を用いて，より大規模なプログラムに対して，さらに多くの実行時ノードの評価を間違えたと指摘が行われる可能性がある．しかし，この指摘はプログラマが評価を行った実行時ノードにのみされるものであり，そもそもプログラマが自らが把握できないほど大量な評価をするとは考えにくい．さらに，この指摘はあくまで，故障している可能性を秘めていることを示しているので，実際には故障していなくても，これらの情報を指摘して他の実行時ノードにも欠陥が伝播している可能性があることをプログラマに示すことは有用であると考えられる．

4.3 組合せ最適化問題の解の数について

この規模の大きさのプログラムであっても，表 2 の 1 行あたり 12 個の最適解が求まった．このことから考えて，さらに大きなプログラムを扱う場合は最適解の数が増すことが考えられるので，その中からさらに適切なものを選択するアルゴリズムが必要になる．そのアルゴリズムは，欠陥として指摘するノード数が少ないものを優先することが良いと考えられるが，さらに研究の必要がある．

4.4 計算量について

本手法で用いた組合せ最適化問題は，組合せ爆発が発生してしまい短時間で効率よく解くことはできない問題である．そのため，効率よく処理を行うための枝刈り法を考える必要がある．考えられる解決策としては，次のことが考えられる

- 幅優先探索によって最適解を求める．
- システムが機械的に割り当てる誤ラベルの上限を決めてしまう．
- 欠陥が含まれていそうな範囲をプログラマにある程度絞ってもらう．

また，今回の事例実験では，準最適解であっても，妥当な結果が得られそうな感覚を得た．これらを組合せてシステム化してみることが今後の課題である．

5. おわりに

本稿では，プログラマが持つ思い込みを指摘できるシステムの構想と机上の実験に基づく実現可能性について述べた．本手法ではプログラムの実行過程とプログラム作成時のプログラマの意図の両方を表現した結合グラフを生成する．次に，結合グラフを対象に，プログラマがわかる範囲で意図通り動作しているか否かの評価を行って，該当するノードに正/誤のラベルをつける．その次に，プログラマの評価の情報を最も上手く説明ができる欠陥の位置を，組合せ最適化問題にして解き，探し出す．その際同時に，最適なラベル割り当てと，プログラマの評価が食い違っているノードが発生するが，この食い違ったノードに関してプログラマが思い込みをしている可能性をシステムが指摘する．この指摘によって，プログラマがなんらかの思い込みを持っていたことを気づかせることができる．

小規模な事例実験の結果，プログラマが自分の書いたプログラムは自分の意図していることを正確に表現していると思い込むことを，13 個の事例の全てで指摘することができた．また，プログラマがプログラム実行の様子を観察している際に，自らの意図していたものとは異なる動作をしているにもかかわらず，これを意図通りに動いていると思い込むこと（もしくは意図通りに動作しているにもかかわらず，これを意図通りに動いていないと思い込むこと）に対する指摘は，13 個の事例の内 8 個指摘することができた．

本手法の一番の課題は計算量の問題である．如何に組み合わせ爆発を抑え，効率よく処理を行うかでこの手法を用いたシステムの実現性が大きく左右されると考えられ

る．また，本手法はまだ机上実験のみが行われた状態なので，今後実際にシステムを作成し，より多くの実験を行う必要があると考えられる．

参考文献

1) 石井康雄（編）：ソフトウェアの製造，日科技連ソフトウェア品質管理シリーズ，第 3 巻，日科技連，1986.
2) E.Shapiro: Algorithmic Program Debugging, Ph.D. Thesis, MIT Press (1982).
3) P.Fritzson, N.Shahmehri, M.Kamkar, and T.Gyimothy: Generalized Algorithmic Debugging and Testing, ACM Letters on Programming Languages and Systems, vol.1, no.4, pp.303−322 (1992).
4) M.Weiser: Program Slicing, IEEE Trans. on Softw. Eng., vol. 10, no.4, pp.352−357 (1984).
5) B.Korel and J.Laski: Dynamic Slicing of Computer Programs, The Journal of Systems and Software, vol.13, no.3, pp.187−195 (1990).
6) H.Agrawal and J.Horgan: Dynamic Program Slicing, *ACM SIGPLAN Notices*, vo.25, no.6, pp. 246−256 (1990).
7) A.Zeller and R.Hildebrandt: Simplifying and Isolating Failure-Inducing Input, IEEE Trans. on Softw. Eng., vol.28, no.2, pp.183−200 (2002).
8) A.Ko and B.Myers: Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior, *Proc. of the 2004 Conf. on Human Factors in Computing Systems*, pp.151−158,Vienna (2004).
9) M.Ernst, J.Cockrell, W.Griswold, and D.Notkin: Dynamically Discovering Likely Program Invariants to Support Program Evolution, IEEE Trans. on Softw. Eng., vol.27, no.2 , pp.1−25 (2001).
10) S.Hangal and M.Lam: Tracking Down Software Bugs Using Automatic Anomaly Detection, Proc. of 24th Intl. Conf. on Softw. Eng., pp. 291−302 (2002).
11) A.Groce and W.Visser: What Went Wrong: Explaining Counterexamples, Proc. of the SPIN Workshop on Model Checking of Software, pp. 121−135 (2003).
12) A.Zeller: Why programs fail – A guide to systematic debugging (2nd ed.), Morgan Kaufmann, 2009.
13) G.Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. Visaggio: Evaluating performances of pair designing in industry, The Journal of Systems and Software, vol.80, pp.1317–1327 (2007).