ソフトウェア工学 69 − 6
(1 9 8 9. 1 1. 2 4)

プログラムのデバッグに対する知的支援

山田宏之　相原恒博
愛媛大学　工学部

プログラム開発において，プログラムの効率のよいデバッグを実現するには多種多様のプログラミングに関する知識とデバッグに対する経験を必要とするために，プログラマにとって，デバッグは容易な作業ではない．そこで，本稿ではプログラムのバグをプログラム検証の手法を用いて検出し，バグ修正に対するアドバイスをユーザに与えることにより，デバッグを支援するシステムの構成に関する基礎的な考察を行う．本システムが対象とするプログラムは制限付きのＬＩＳＰ言語で再帰的に定義された関数である．本システムには，プログラムの正しい仕様を表現する参照プログラムとバグを含む可能性のある検証対象プログラムとの対が入力され，各々グラフ表現（プログラムグラフ）に変換される．つぎに，２つのプログラムから検証条件が作成され，２つのプログラムの等価性が検証される．検証が成功する場合は，２つのプログラムは等価であることが証明され，検証対象プログラムにバグは含まれていない．一方，検証に失敗した場合には，プログラムにバグがあることが判明し，検証に失敗した検証条件からバグを同定し，バグの修正案を作成し，ユーザに提示する．この修正案は参照プログラムから生成される．

Intelligent assistance for debugging a program

Hiroyuki YAMADA and Tsunehiro AIBARA
Ehime Univ.
3, Bunkyo-cho, Matsuyama, Ehime 790, Japan

It is very difficult for a programmer to correct a buggy program efficiently. This paper presents fundamental consideration about an implementation of an intelligent assisting system for debugging a program defined recursively in restricted LISP. Our system can detect bugs in the program by program verification techniques. Moreover, it can make advice on how to fix the bugs in order to assist him to correct it. Our system inputs a pair of a correct program (a reference program) and a buggy program (a target program). The reference program is regarded as specification. Then, the verification conditions are generated from the programs and these conditions are checked. This system verifies the computational equivalence between the reference program and the target program. If the verification is failure, bugs are detected. Then, an example of the bug correction is generated from the reference program.

１．はじめに

プログラム開発において，プログラムの保守は重要な作業の一つである．その中でも，プログラムのデバッグは，問題解決法に関する知識，プログラミング技法に関する知識，プログラミング言語に関する知識およびデバッグに対する経験的知識等のさまざまな知識を必要とするために，プログラム作成と同様に，非常に高度な技能をプログラマに要求する．その結果，プログラミングの経験の少ないプログラマをはじめとして，多くのプログラマはデバッグに多大な労力と時間を費やしており，プログラムの開発効率を改善するには，デバッグを適切に支援する必要がある．

近年，プログラム開発の効率を向上させるために，知識工学で得られた成果を応用して，プログラミング作業をあらゆる面から知的に支援する知的プログラミング環境の構築に関する研究がなされている[1],[2]．この知的プログラミング環境に関する研究の中に，初心者が開発したプログラムに含まれるバグを検出し，バグが発生した理由およびバグの訂正法を示すことにより，プログラムのデバッグ支援に加えて，初心者のプログラミング教育も同時に実現することをめざす知的支援システムの開発がなされている[3]-[5]．しかしながら，これらのシステムは，デバッグできる状況が限定されており，十分な成果は得られてない．

従来から，プログラムの正しさを検査する手法として，プログラムの検証法が考察されている[6]．プログラムの検証とは，開発されたプログラムがプログラムの満足すべき性質を記述した仕様を満足していることを検査することである．したがって，仕様が正しいと仮定すれば，検証に失敗する場合には，プログラム内にバグがあることが判明する．

そこで，本稿ではプログラム検証の手法を利用して，プログラムに含まれるバグを検出し，バグの修正案を生成する手法について考察する．本システムでは，デバッグ対象のプログラムを記述する言語をＬＩＳＰ言語のサブセットとし，この言語により再帰的に定義されたプログラムを扱う．

以下においては，まず，プログラムの自動デバッグを実現する上での問題点を明確にし，デバッグ時にユーザの作業を支援する知的デバッグ支援システムが備えるべき機能について考察する．つぎに，２つのプログラムの等価性を検査するプログラム検証によりプログラム内のバグを検出し，バグの修正案を提示するシステムについて述べる．最後に，具体例をとおして本稿で用いている手法によりプログラム内のバグを検出できることを示す．

２．自動デバッグの問題点

２．１　プログラムの多様性

プログラムに対するデバッグの自動化を困難にする要因の一つに，プログラムの実現には非常に多くの自由度をもつことが挙げられる．ここで，プログラムの自由度とは，ある与えられた問題を解決するためのプログラムの正しい実現が一意に定まらないことを指している．例えば，同じ問題を解決するプログラムには，以下に示すような自由度があるために，実現されたプログラムには無数のバリエーションがある．

（１）問題解決法の自由度

ある問題が与えられたとき，それを解くための方法が複数存在する．例えば，列べ換え（ソーティング）の問題では，クイックソート，バブルソート，ヒープソート等の解き方があり，いずれを採用しても与えられた問題を解決できる．

（２）コーディングの自由度

同じ問題解決法を実現しているプログラムが複数存在する場合に，以下に述べるような自由度のために，それらが全く同じコーディングで実現されていることは非常に少ない．

①　同じアルゴリズムを異なった構造で実現できる．例えば，階乗を求めるプログラムにおいて，その繰り返しを再帰的に記述することもでき，ループ構造で記述することもできる．

②　同じ機能を異なったプログラミングプリミティブを用いて実現できる．例えば，反復を実現する場合に，ＬＯＯＰによる方法，ＷＨＩＬＥによる方法がある．

③　プログラム内の識別子は同じ名前でも役割が異なったり，異なる名前でも同じ役割をもったりすることがある．

④　同じ機能をもつ関数でも，仮引数の順番が異なる場合がある．

プログラムのデバッグを支援するシステムは，上述のような正しいプログラムの無数のバリエーションに対処する必要がある．

２．２　多種多様な無数のバグ発生

プログラムのデバッグを困難にする別の要因に，プログラム中に多種多様な，無数のバグが発生する可能性のあることがある．たとえば，以下に述べる状況でバグが容易に発生する．

（１）入力ミス

これは，識別子等のタイプミスやスペルミスにより発生するバグである．このバグは比較的容易に認識でき，修正できる．

（２）　変数に対する更新場所の間違い

これは，変数を更新する場所を誤ったり，更新を忘れたりするために，発生するバグである．

（３）誤った条件分岐

これは，問題を分析するときに考慮すべき条件分岐の場合分けを見落としたり，誤ったりするために，発生するバグである．

（４）予期しない副作用

これは，プログラミングプリミティブ間の相互作用により発生するバグである．プログラムの個々の部分は正しく見えるが，全体としては正しくない場合がこれに相当する．

これらの状況（ただし，（１）は除く）でのバグは，プログラマの勘違いやプログラミングに関する知識あるいは問題解決法に関する知識の欠如により発生すると考えられる．いずれの場合でも，プログラマは現在の状況を正しく認識できないときには，プログラムを正しくデバッグできない状況に陥ることがある．このような状況で，システムが適切な助言を与えることは意義が大きい．

２．３　デバッグ支援に要求される機能

先に述べた問題点に対処し，デバッグ作業をあらゆる点から支援できるシステムを構築する上で，システムに要求される機能には以下のものがある．

（１）プログラムの意味を理解できる機能

デバッグ対象のプログラムのコードが何をするかをコードから推論できることは，プログラムのもつ自由度に対処する能力を実現できる．プログラムの理解には複数のレベルがある[1]．

（２）ユーザの意図を理解できる機能

ユーザの意図がプログラムのソースコードから推論できることは，プログラムの意味的なバグ検出の能力をシステムにもたせるために必要な機能である．

（３）ユーザの技能レベルを理解できる機能

ユーザのプログラミングに対する技能レベル，頻繁におこす誤り，思い違い等のユーザに関する情報を推論できることは，より適切な忠告を与えることができる等，高度な支援を実現する上で必要である．この機能はプログラミング教育をコンピュータにより行う知的チュータシステムにとっても重要な機能である．

本稿では，システムが２つの等価なプログラムを正しく等価であると識別できるならば，そのシステムはプログラムの意味を理解しているとして，そのための機構をプログラム検証により実現する．ただし，(2)，（３）については，本稿では考察していない．

３．デバッグ支援システム

デバッグ支援システムの構成を図１に示す．システ

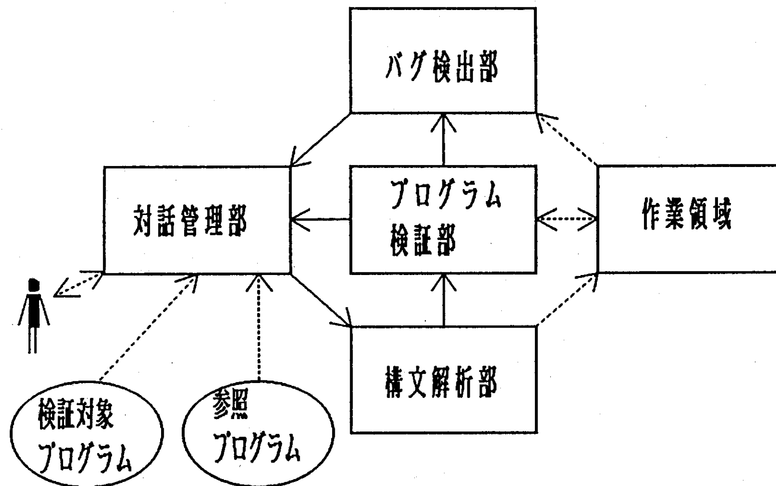

図１．システム構成

ムにはデバッグの対象となるプログラム（検証対象プログラム）と検証対象プログラムの仕様に相当するプログラム（参照プログラム）との対を入力する．ここで，参照プログラムは，検証対象プログラムと同じアルゴリズムで正しく実現されたプログラムである．それぞれのプログラムは構文解析部により構文解析され，内部表現（プログラムグラフ）に変換され，作業領域に蓄えられる．つぎに，プログラム検証部において，プログラムグラフに基づき，検証条件が作成され，2つのプログラムの計算的な等価性が検証される．検証に失敗した場合には，バグ検出部により検証条件から検証対象プログラムのバグを検出し，参照プログラムを利用して，バグの修正案を生成し，対話管理部をとおしてユーザに提示する．

以下では，本システムが対象とするプログラムを記述する言語に与えた制約を説明し，個々のモジュールの機能を述べる．

3．1　対象プログラム記述言語

本システムでは，対象プログラム記述言語として，ＬＩＳＰ言語のサブセットを採用している．この言語は，リスト処理関数（car, cdr, cons等），条件分岐関数（cond, if），算術演算用関数（+, -等），述語関数（zerop, null, listp等）から構成され，副作用を伴う関数（setq, setf等）は含まない．したがって，現段階では副作用のないプログラムを対象とする．

また，上述の言語で記述されるプログラム（関数）は再帰的に定義されるものとし，プログラム内からは他のユーザ定義関数を呼び出すことはできない．つまり，本デバッグ支援システムがデバッグできるプログラムのレベルは，ＬＩＳＰの入門書[7]の最初に現れる初歩的な例題レベルである．

将来的には，扱えるプログラムのレベルを向上させるつもりである．

3．2　構文解析部

構文解析部では，検証対象プログラムおよび参照プログラムを，条件分岐に基づいて，グラフ構造（プログラムグラフ）に変換する．プログラムグラフは，ノードとノード間を結ぶリンクとにより構成される．ノードには，中間ノードと終端ノードの2種類のノードがある．中間ノードは条件分岐を表現し，ノード内に分岐条件が記述される．また，終端ノードには条件分岐に関する記述を含まないプログラムコードの断片が記述される．中間ノードがもつ条件が成立する場合には，Ｔとラベルづけされたリンクがたどられ，条件が成立しない場合には，Ｆとラベルづけされたリンクがたどられる．

例えば，図2に示すプログラムのプログラムグラフを図3に示す．このプログラムは，新しいセルを用いて，入力されたリストのコピーを生成するものである．図3において，変数ｌがアトムならば左側のノードが有効になり，アトムでなければ右側のノードが有効になる．したがって，プログラムの実行は，そのルートから終端ノードまでグラフをたどり，その終端ノードに到達するまでに通過した中間ノードにより表現される条件のもとで，終端ノードがもつプログラムコードを実行することに相当する．

3．3　プログラム検証部

プログラム検証部では，検証対象プログラムと参照プログラムとから検証条件を生成し，検証対象プログラムが参照プログラムと計算的に等価であることおよび検証対象プログラムが再帰呼び出しの結果，停止することを検証する．

まず，検証対象プログラムのプログラムグラフと参照プログラムのプログラムグラフ間で同じ条件で到達する終端ノードのペアを生成する．

つぎに，生成された終端ノードのペアから検証条件を生成する．検証条件は，終端ノードに到達するまでの分岐条件（仮説）とその仮説が成立するときに各終端ノードに記述されているプログラムコードの断片が計算的に等価であることの記述とから構成される．プ

```
(defun copy (l)
   (if (atom l) l
       (cons (copy (car l)) (copy (cdr l)))))
```

図2．　参照プログラム（ｃｏｐｙ）

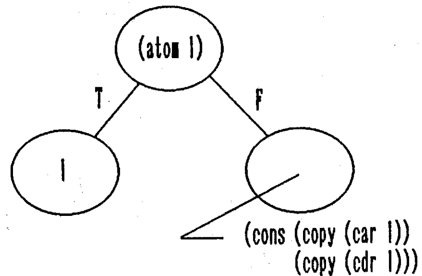

図3　参照プログラムのプログラムグラフ

ログラムの検証は，２つのステップで実行される．まず，検証条件の仮説が成立するときに，２つのプログラムコードの断片が等価であるか調べる．つぎに，等価であることが示されたときには，検証対象プログラムのコードの断片内に再帰呼び出しが含まれているか調べる．もし，含まれている場合には，再帰呼び出しに含まれている引数の一つに着目し，その再帰呼び出しが実行される毎に，必ずその引数がもつ値の大きさ（リストの長さ，数の大きさ等）が減少することを検査する．もし，大きさが減少するならば，その検証は成功し，検証対象プログラムのコードの断片は正しいと判断される．すなわち，そのコードの断片にはバグが含まれていないと判断する．しかしながら，大きさが減少しない場合，あるいは，２つのコードの断片が計算的に等価でない場合には，検証対象プログラムのコードの断片に何等かのバグが含まれていると判断される．

３．４　バグ検出部

バグ検出部では，まず，検証に失敗した検証条件より，検証対象プログラムに含まれるバグを同定し，修正候補を生成する．具体的には，検証条件に含まれる検証対象プログラムのコードと参照プログラムのコードを比較し，最小の差異を抽出する．この差異の置換をバグ修正候補とする．さらに，生成された修正候補に基づき新たな検証条件を生成する．検証条件が成立することを検証し，検証に成功した場合には，その修正候補を修正案としてユーザに示す．

３．５　実行例

ここでは，具体例を通して本システムの動作を説明する．ユーザが３．２で述べた例題（リストのコピーを生成する関数）に対して図４に示すようなバグを含むプログラムを作成したものとする．図４において下線で示されている部分がバグである．このプログラムのプログラムグラフを図５に示す．

まず，参照プログラムのプログラムグラフ（図３）と検証対象プログラムのプログラムグラフ（図５）間で，条件の場合分けを比較し，ノード間の対応関係を明確にする．このとき，もし，参照プログラムにはあるが，検証対象プログラムにはない場合分けが発見されたならば，条件分岐の欠落がユーザに指摘される．

つぎに，対応関係に基づいて検証条件を生成する．参照プログラムから，分岐条件に基づき場合分けを行う．この場合には，(atom l)が成立する場合と成立しない場合に場合分けできることが分かる．さらに，前者の場合を，検証対象プログラムから，(null l)が成立する場合と成立しない場合とに分ける．その結果，この例題の場合には以下に示す３つが検証条件として生成される．

```
(-> (and (atom l) (null l))                (1)
    (equal l nil) )
(-> (and (atom l) (not (null l)))          (2)
    (equal l (list l)) )
(-> (not (atom l))                         (3)
    (equal (cons (strange (car l))
                 (strange (cdr l))))
    (equal (cons (strange l)
                 (strange (cdr l)))) )
```

検証は以下のように進められる．

まず，システムは検証条件 (1)が成立するかどうかを調べる．変数ｌの値がアトムで，かつ，ナルならば，その値とｎｉｌとは等価であるかどうか調べる．ここで，検証条件 (1)は明らかに成立する．

つぎに，変数ｌの値がアトムで，かつ，ナルでないときに，その値とその値のリストが等価であるかどうか調べる．この場合に検証条件 (2)は，明らかに成立しない．そこで，バグ検出部にこの検証条件が渡され，バグの検出を行う．バグの検出は，検証対象プログラムのコードと参照プログラムのコードを比較し，相違

```
(defun strange (l)
  (cond ((null l) nil)
        ((not (listp l)) (list l))
        (t (cons (strange l)
                 (strange (cdr l))))))
```

図４．検証対象プログラム

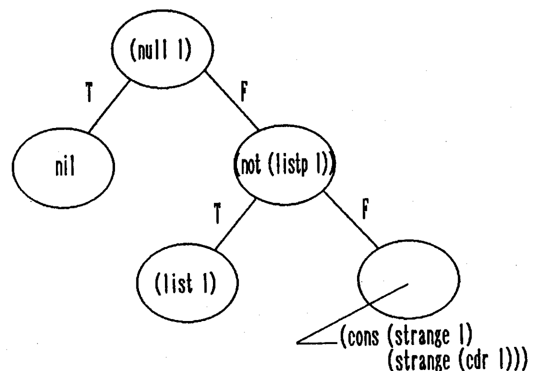

図５．　検証対象プログラムのプログラムグラフ

点をバグ候補として選ぶ．この場合には(list 1)がバグの候補として選ばれる．

最後に，検証条件（3)を証明するために，1がアトムでないときに，システムは，参照プログラムの

(cons (strange (car l)) (strange (cdr l))))

と，検証対象プログラムの

(cons (strange l) (strange (cdr l))))

とが等価であることを証明することを試みる．しかしながら，この2つのコードは等価でないために，この検証には失敗する．そこで，この検証条件をバグ検出部に渡し，バグの検出を試みる．コード間の比較により，参照プログラムのコード（strange (car l))と検証対象プログラムのコード（strange l)とが異なることが検出される．そこで，検証条件に含まれている後者のコードを前者のコードと置き換えて再度検証を試みる．その結果，今度は検証に成功し，システムがこの2つのコードの差異を調べ，最終的に（car l)と l とをコード間の差異として検出する．その結果，検査対象プログラム内の最初の l がバグであることが指摘される．

以上の検証を通して，検証対象プログラマ内のバグはすべて検出される．

4．本実験システムの評価

本稿で述べたデバッグ支援システムの特徴を以下に挙げる．

（1）本システムでは，対象プログラムに対する仕様として正しく実現されたプログラムのコードを利用している．検証は，バグを含む可能性のあるプログラムが参照プログラムと等価であることを検証することにより実現される．

（2）プログラムをグラフ表現することにより，プログラミング言語への依存度を小さくできるので，プログラムの多様性解消に有効であると考えられる．

（3）同じ機能を別のプログラミングプリミティブで構成している場合でも，プログラムコード間の計算的な等価性を検証することにより，プログラムを正しく認識することができる．

（4）プログラマの意図認識および高度なプログラム理解機能がないために，検出できないバグがある．たとえば，検査対象プログラムの最後のコードが

(cons (car l) (strange (cdr l))))

である場合には，同じ入力に対して参照プログラムと同じようにみえる結果を出力するが，プログラマの目的であるリストのコピーを生成するという意味では誤っている．このバグを検出するには，プログラマが何をしたいかをシステムが知ること，および，よりレベルの高いプログラム理解が必要である．

（5）現段階では，対象プログラム記述言語および対象プログラムに課した制約がきびしすぎるために，実用的でない．本稿の結果を踏まえて，制約を緩めるつもりである．

5．むすび

本稿では，プログラム内のバグをプログラム検証の手法を用いて検出する知的デバッグ支援システムの構成に関する基礎的考察を行った．

本システムにより，簡単な例題ではあるが，プログラム中のバグの検出ができることを示した．

本システムは現在構築中であり，検証できるプログラムにかなり制約がある．そこで，今後の課題としては，プログラム記述言語の制約を緩めること，よりバグ検出能力を向上させるために，プログラマの意図を推論する機構を考察することが挙げられる．

[謝　辞]

日頃，御激励を頂く大阪大学工学部の手塚慶一教授に感謝の意を表す．

[参考文献]

[1] 上野：" 知的プログラミング環境ープログラム理解を中心にー"，情報処理，Vol.28，No.10，pp.1280-1296，1987.

[2] Waters,R.C. : "The Programmer's Apprentice: Knowledge Based Program Editing", IEEE Trans. on SE., vol. SE-8, pp.1-12, 1982.

[3] Johnson, W.L. and Soloway, E. : "PROUST: Knowledge-Based Program Understanding", IEEE Trans. on SE., Vol.SE-11, No.3, pp.267-275, 1985.

[4] Adam,A. and Laurent,J. : "LAURA, A System to Debug Student Programs", Artifitial Intelligence, 15, pp.75-122, 1980.

[5] Murray, W. R. : "Automatic program debugging for intelligent tutoring systems", Pitman, 1988.

[6] 斉藤，米崎："ソフトウェアの検証"，ソフトウェア工学ハンドブック，オーム社，pp.177-221 (1986).

[7] Winston, P.H. and Horn, B.K.P : "LISP", 3rd Edition, Addison Wesley, 1989.

[8] Fitting,M. : "Computability Theory, Semantics, and Logic Programming", Oxford Univ. Press, 1987.