CASE ツールの開発におけるソフトウェアバグの分析

下村 隆夫†

ソフトウェア開発時に検出したバグを分析することによって，バグの潜在箇所，バグ潜在確率，バグを引き起こす原因等を明らかにし，それらに基づいてバグの防止策を提案する．また，これらのバグ分析結果の具体的な応用について考察する．検出したバグの分析は，「エラー発見状況」，「バグ究明手順」，「バグ修正箇所」，「バグの原因/防止策」等を記入できる帳票（バグ分析票）を用意し，このバグ分析票を用いて行った．バグ原因の分析に基づいて提案したバグ防止策を実践することにより，約25%のバグを未然に防止できる可能性があることが明らかとなった．

Analysis of Software Bugs in Developing CASE Tools

TAKAO SHIMOMURA†

By analyzing the bugs detected during software development, this paper makes it clear where bugs exist, the probability that they exist there, and what causes them, and as a result, presents several ways of preventing bugs from being included in programs. The paper finally discusses some concrete applications of these results. The detected bugs were analyzed using a bug-analysis form in which an error state, the procedure of having located a bug, corrected code, bug causes, and an idea to prevent it can be written. It has been made clear that by applying the ways of preventing bugs this paper presents, the number of bugs can be reduced by twenty-five percent before proceeding to a program-testing phase.

1. はじめに

ソフトウェアバグについては，バグ検出能力に関する評価，残存バグ数の予測，バグの影響範囲に関する分析等.いくつかの実験結果が報告されている,バグ検出能力に関しては，Howden[1),2)]，Gannon[3)]，Sneed[4)]では，ブランチテスト，パステスト，データフローテスト，同値分割法，コードインスペクション，静的解析ツール等の各種テスト技法のバグ検出能力を検出したバグ数により比較している．残存バグ数の予測については，Fitzsimmons[5)]ではプログラムの構文上の特性から，Nakagawa[6)]では，テストによって検出されるエラーの複雑度の変化から残存バグ数を予測している．バグの影響範囲に関しては，Endres[7)]では，結合テストであるにも関わらず，バグの85%は1つのモジュールの修正で吸収できたことを報告している．また，バグの原因を，問題を正しく理解しなかったことによるバグ（46%），プログラムコード作成上のバグ（38%），環境等のエラー（16%）に大きく分類している．これらの研究では，検出バグ数が関心の対象となっており，バグの内容に関する詳細な分析はされていない．

Mohri[8)]では，Jackson Structured Programmingを用いたソフトウェア開発において，バグ存在箇所，バグ修正プロダクトやバグ原因等から，バグが混入した開発ステップを識別することができるバグ分析手続きを提案している．早い段階でバグを除去することが重要であり，そのためには，レビューが有効であることを述べているが，レビューにおけるバグの除去方法については言及していない．

これに対し，本論文では，ソフトウェア開発時に検出したバグを分析することによって，バグの潜在している箇所，バグの潜在している確率，バグを引き起こす原因等を明らかにし，それらに基づいてバグの防止策を提案する．日野[9)]では，バグを分析することにより，まず，具体的なバグ防止策を提案し，次に，それを，単純，一様，対称，階層，透過，明証，安全という一般的なプログラム設計/作成原理にまとめている．本論文で述べるバグの防止策とは互いに補完し合うものである．また，本論文では，このほかに，文種別ごとのバグの分布，バグ発生確率，バグ究明時間の分布，

† ATR 通信システム研究所
ATR Communication Systems Research Laboratories

および，文種別とバグ究明時間との間の相関関係等について分析を行い，最後に，これらの分析結果の応用方法について述べる．

検出したバグの分析は，「エラー発見状況」，「バグ究明手順」，「バグ修正箇所」，「バグの原因/防止策」等を記入できる帳票（バグ分析票）を用意し，このバグ分析票を用いて行った．バグ原因の分析に基づいて提案したバグ防止策を実践することにより，約25%のバグを未然に防止できる可能性があることが明らかとなった．

本論文では，まず，2章でバグ分析票に基づいたバグの分析方法について述べ，3章でエラーの発生比率，バグ究明時間の分布，バグの存在箇所，バグ発生確率，エラー種別/文種別/バグ種別の関係，および，それら要因のバグ究明時間に与える影響等について分析を行う．また，4章ではバグの原因を分析し，バグを未然に防止する方法を提案する．最後に，5章ではこれらのバグ分析結果の具体的な応用方法について考察する．

2. バグ分析方法

2.1 分析対象ソフトウェア

バグ分析の対象としたソフトウェアは，SUN ワークステーション上のウィンドウ環境で動作するC言語で記述されたソフトウェア開発支援ツール群（エディタ，レポジトリ，デバッガ等）である．ソフトウェアの開発は4社に発注して行った．

開発時期は次の2つに分かれている．

[EXPR 1]

A社，B社，C社の3社，合計88 KS.

分析対象としたバグは363件.

[EXPR 2]

D社50 KS.

分析対象としたバグは215件.

2.2 バグ分析票

分析の対象としたバグは，ソフトウェア開発の結合試験工程で検出したバグである．「エラー発見状況」，「バグ究明手順」，「バグ修正箇所」，「バグの原因/防止策」等を記入できる帳票（バグ分析票）を用意し，このバグ分析票を用いて検出したバグの分析を行った．バグ分析票のフォーマットを**表1**に示す．

「エラー発見状況」欄はテスト担当者が，「バグ究明手順」欄と「バグ修正箇所」欄はデバッグ担当者が，「バグの原因/防止策」欄はコーディング担当者が記入した．バグ分析票の各欄に記述例を載せることにより，記入が容易になるように工夫してある．イタリック体で表されている部分は，あるバグについて実際にバグ分析票に記入した例を示している．各記述欄の詳細を以下に示す．

[エラーの発見]

1) エラー状況

プログラムをテストしたときに発見された誤り（以下，エラーと呼ぶ）の状況を記述する．

2) エラー種別

プログラムテストによって観察されるエラーを以下のように分類した．この中，「その他」に属するバグは実際のバグではないため，分析の対象からは除外した．

- 出力異常
  出力/表示された内容が誤っている．
- 異常終了
  アボートし，強制終了となった．
- 処理漏れ
  ある処理が実行されない．
- 応答なし
  応答がなく，処理を継続できない．
- その他
  実行環境の誤り（例：入力ファイルが存在しない）等である．

[バグの究明]

3) バグ究明手順

エラーを発見した後，バグを検出するために行ったバグ究明の手順を箇条書で記述する．

4) バグ究明時間

エラーを発見した後，バグを検出するためにかかった時間を記述する．「机上」は，ソースリストを見て調べた時間，「マシン」はマシンを使用してバグの究明にかけた時間を表す．

[バグの修正]

5) バグ修正内容

バグ修正前とバグ修正後のソースコードを記述する．修正部のソースリストを添付してもよい．

6) 文種別

バグを含む文の種別（分岐文/ループ文の条件式，関数呼出し文，代入文，宣言文，その他）を記述する．

複数の種類の文にまたがって修正がある場合（組み合わせ）は，「その他」とした．

表 1 バグ分析票
Table 1 Bug analysis form.

1. エラーの発見	エラーの状態を記述する. [テスト担当：小川]	
	例 1. 応答がなく，キー入力も効かない. 例 2. アボートし，コアダンプした. エラー状態：動的にアロケートされるリスト構造が複数存在すると， データフロー表示において反転表示する領域がずれてしまう.	エラー種別 ○出力異常 異常終了 処理漏れ 応答なし その他
2. バグの究明	バグ箇所を究明した手順を記述する. [デバッグ担当：斎藤]	
	例 1. ①トレースを入れて再実行し，実行到達範囲を限定した. ② 机上で，実行到達点を含むループ文を調べた. 例 2. ① トレースを入れて再実行し，アボートした文を限定した. ② プローブを入れて再実行し，アボートした文を含む関数のパラメータ値を調べた. バグ究明手順：① トレースを入れて再実行し，X座標が不正な値となる箇所を調べた. ② winvar( ) からのリターン時にX座標が不正な値となっていることを確認した. ③ winvar( ) の内部にトレースを入れて，不正な値を設定している箇所を調べた. ④	バグ究明時間 時間 30分 机上 時間 0分 マシン 時間 30分
3. バグの修正	バグの修正前，および，修正後のソースコードを記述する. [デバッグ担当：斎藤]	
	例 1. while(flag=1) →while(flag==1) 例 2. tTable *tableptr; Initialize_Table(tableptr); →tTable table; Initialize_Table(&table); 修正前：*x=winw1+WINSW1+listtree[i].column*(lfigw+LSPACEH); → 修正後：*x=winw1+WINSW1+listtee[i].column*(lfigw+LSPACEW);	文種別 分岐文 関数 call ○代入文 宣言文 その他 (組合せ) バグ種別 文の漏れ ○文の誤り 文の過多
4. バグ原因とバグ防止策	バグの原因とバグ防止策を記述する. [コーディング担当：斎藤]	
	例 1. 条件式で==とすべきところを代入文＝としてしまった. →机上レビュー時に，＝を含む条件式を全て確認しておく. 例 2. 値の未定義なポインタがアーギュメントとして関数に渡された. →予め，領域を取って，そのアドレスを渡す旨を当該関数のプログラム仕様書に明記しておく. バグ原因：計算式に用いるシンボル名を間違えていた. → バグ防止策：コードレビュー時に不注意なコーディングミスがないかチェックする.	

7) バグ種別

バグの種別を以下のように分類した.

- 文の漏れ
- 文の誤り
- 文の過多（余分な文）

これらの組み合わせは「文の誤り」とした.

[バグ原因とバグ防止策]

8) バグ原因

どのような不注意，勘違い等のためにバグを引き起こしたかを記述する.

9) バグ防止策

そのような不注意，勘違いを未然に防ぐとしたら，どのような方法，工夫があるかを記述する.

3. バグの分析

EXPR 1 のバグ分析データを用いて，エラー種別ごとの発生比率，バグ究明手順，バグ究明時間の分布，バグの存在箇所，バグの防止策等に関する分析を行った．EXPR 2 では，さらに，エラー種別/バグ種別/文種別間の相関関係を解析するため，バグ分析票に「エラー種別」「バグ種別」「文種別」欄を付け加えて，バグの記録，分析を行った．また，これらの要因のバグ究明時間に与える影響についても解析を行った.

3.1 エラー種別ごとの発生比率

エラー種別ごとの発生比率を**図1**に示す．エラーの多くは出力異常（62%）という形で発見されていることがわかる.

3.2 バグ究明手順

エラーを発見した後，バグを検出するために行ったバグ究明の手順は，エラー種別ごとに，概ね次のように分類できた．バグ究明時間の長いものは，プローブを入れてプログラムを再実行し，いろいろな情報を調べている場合が多い.

（a）出力異常

誤っている出力/表示内容を見て，それに関係している関数がわかる場合が多い．その関数の処理内容を机上あるいはマシンで調べることにより，バグを検出している．関数が特定できない場合には，関数トレースを入れて再実行し，異常な出力を行った関数を突き止めている.

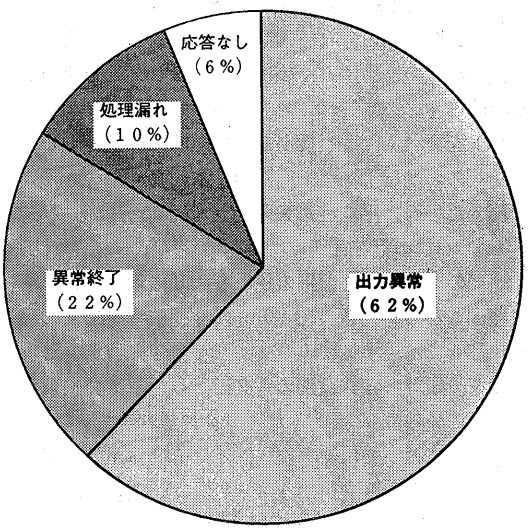

図 1 エラー種別ごとの発生比率
Fig. 1 Frequency ratio of each error type.

（b）異常終了

デバッガを用いて call スタックを調べることにより，あるいは，関数トレースにより，異常終了した関数を特定している.

（c）処理漏れ

漏れていた処理の実行を受け持っている関数を机上あるいはマシンで調べている.

（d）応答なし

関数トレースによりループしている関数を突き止めている.

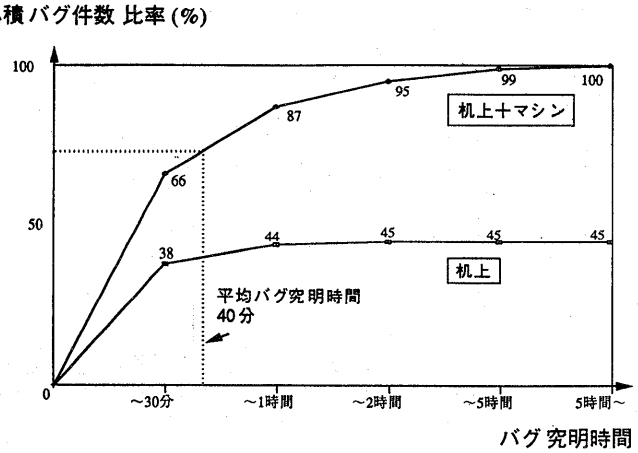

図 2 バグ究明時間の分布
Fig. 2 Distribution of the time required to locate bugs.

3.3 バグ究明時間の分布

バグ究明時間の分布を**図2**に示す．エラーを発見した後，バグを検出するためにかかったバグ究明時間の平均は約40分であった．バグの45%は机上でソースリストを調べるだけで検出できている．机上のみのデバッグにおける平均バグ究明時間は約22分，マシンも使った場合の平均バグ究明時間は約55分であった．

3.4 バグの存在箇所

バグの存在する文種別の比率を**図3**に示す．また，文種別ごとの1文当たりのバグ発生確率を**図4**に示す．図4では，横軸に100文当たりの各文種別の出現文数を，縦軸に各文種別の1文当たりのバグ発生確率を示している．100文からなるプログラムがある場合，各文種別において斜線を施した矩形の面積が，当該文種別でバグを含む文がおよそいくつ存在するかを表し，矩形内の%はそのバグの全体に対する比率を表している．

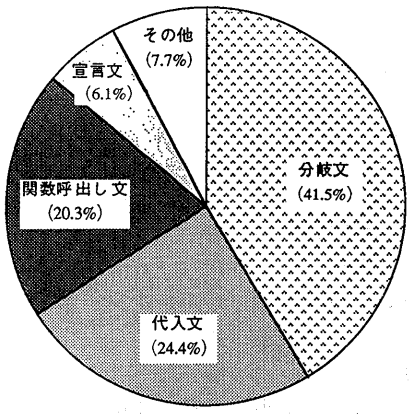

図3 バグの存在する文種別の比率
Fig. 3 Ratio of statement types including bugs.

バグを含む文の種別では，分岐文のバグが最も多く，全体の約42%を占めている．分岐文は1文当たりのバグ発生確率(1.2%)も最も高いことから，コーディングおよびレビュー時には，分岐文の条件漏れ，誤り等のないよう十分に注意する必要があることがわかる．

3.5 相関分析

エラー種別，バグ種別，および，文種別の間の関係をχ^2検定を用いて分析した結果を**図5**に示す．また，これらの要因のバグ究明時間に与える影響を分散分析を用いて分析した結果を**図6**に示す．解析の精度を上げるため，文種別で「その他」に属するバグは解析の対象から除外した．したがって，文種別に関係する解析では，解析対象としたバグ数は169である(EXPR 2におけるバグ総数は215)．

(a) エラー種別とバグ種別との相関

χ^2値の確率は0.0345であり，有意水準5%で有意である．すなわち，エラー種別とバグ種別との関係は独立ではない．図5(a)より，「応答なし」エラーは，「文の漏れ」が原因で発生していることが多いことがわかる．バグ分析票を調べてみると，条件の漏れ，エラー処理の漏れ，メッセージ組み立てが不完全等の原因により，キー入力が効かない，応答がないというエラーが発生していた．

(b) エラー種別の文種別との相関

χ^2値の確率は0.8223であり，エラー種別と文種別との間には有意な依存関係は認められない．

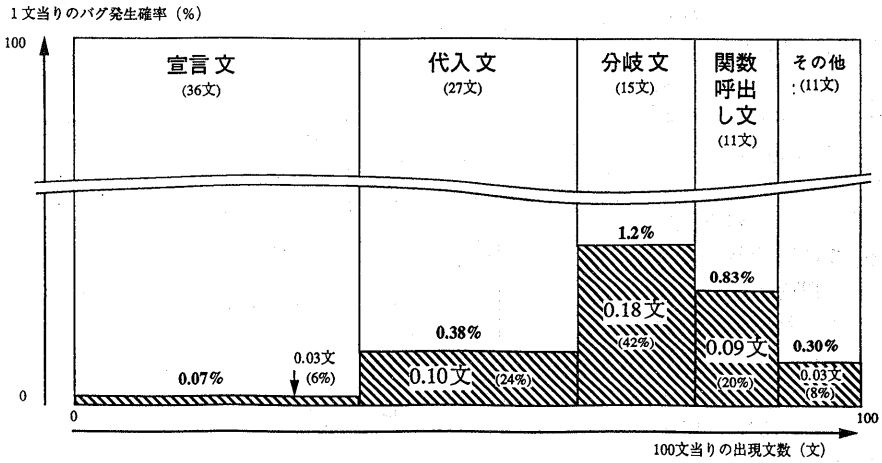

図4 文種別ごとのバグ発生確率
Fig. 4 Probability of bug occurrence according to statement types.

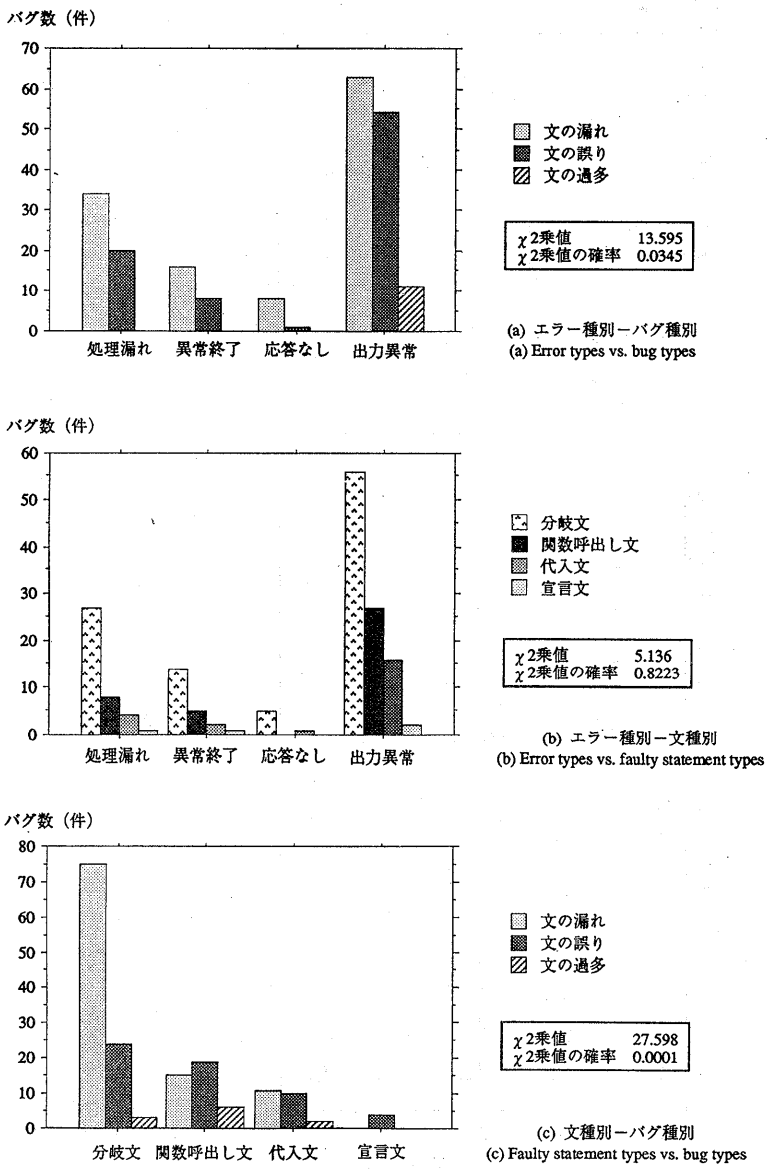

図 5 χ^2 検 定
Fig. 5 Chi-square test.

（c）文種別とバグ種別との相関

χ^2 値の確率は 0.0001 であり，有意水準 1% で有意である．すなわち，文種別とバグ種別との関係は独立ではない．図5（c）より，（1）分岐文には文の漏れが多い，（2）余分な文は関数呼出し文に多いことがわかる．

文の漏れは分岐部分に多いであろうと予想していたが，この仮説は実証された．分岐文は1文当たりのバグ発生確率も高かったことから，分岐文のバグ，特に，分岐文の漏れを防ぐことがバグの減少に大きな効果のあることがわかる．

（d）エラー種別，文種別，バグ種別のバグ究明時間に与える影響

エラー種別とバグ究明時間に関する分散分析における F 値の確率は 0.4402，バグ種別とバグ究明時間に関する分散分析における F 値の確率は 0.9636，文種

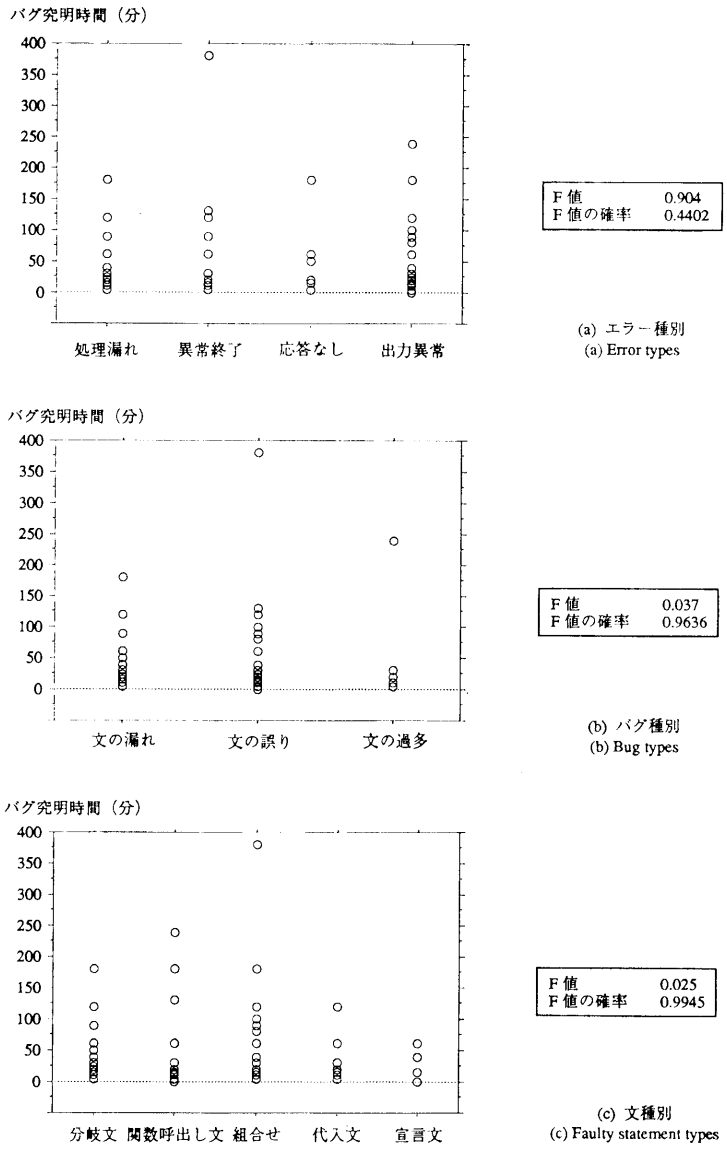

図 6 分散分析
Fig. 6 Analysis of variance.

別とバグ究明時間に関する分散分析におけるF値の確率は0.9945であり，いずれも有意な差は認められない（**図6**参照）．

分岐文のバグは検出しにくいであろうと予想していたが，その仮説は棄却された．また，文の漏れも検出しにくいであろうと予想していたが，その仮説も棄却された．すなわち，エラー種別，文種別，バグ種別のいずれもバグ究明時間を決定する要因とはならないことがわかった．

4. バグ防止方法

バグ分析票に記述されたバグ防止策は，なかには有用なものもあったが，多くは表層的なものであった（バグ分析票にバグの内容を記録しながら，さらにそのバグの防止策まで考える時間的余裕が，各プログラマにはなかったためと思われる）．そこで，「バグ修正箇所」欄や「バグ原因」欄の記述内容を基に，バグ分析票の集計時に改めて各バグについて，その原因を分

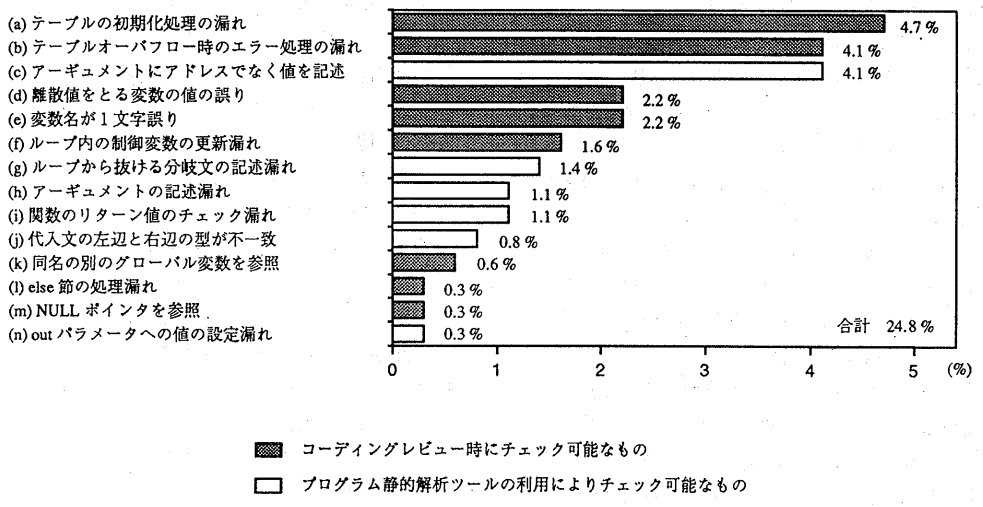

図 7 バグの原因
Fig. 7 Bug causes.

析し，試験工程に入る前にバグを未然に防ぐための可能な方策（すなわち，コーディング時にバグを作り込まない，あるいは，レビュー時にバグを除去するための方策）を検討した．

防止可能と思われるバグの数は，バグ全体 363 件の中の約 25% を占めていた（**図 7**）．図 7 において，(c)，(g)～(j)，および，(n)は，既存のプログラム静的解析ツール（UNIX 上の lint 等）を利用して，型のチェックや制御フロー／データフローの検証等を行うことにより，防止可能と考えられるものである．しかし，既存の静的解析ツールではエラーの一覧が出力されるが，余分なものがあったり，エラーの表現でわかりにくいものがあったりするため，実際の開発では部分的にしか利用されていなかった．エラーを的確に指摘し，エラーの原因や修正方法までガイドしてくれるようなレビュー支援システムが望まれる（5.2 節参照）．

主なバグ防止策を以下に示す．各項番は，図 7 中の項番に対応している．

(a) レビュー時に，すべてのテーブルの各フィールドについて，値を初期設定している処理が存在することを確認する．

(b) テーブルのカウンタ（現在のフィールド位置を指す変数）をインクリメントしている箇所では，テーブルサイズを越えたときのエラー処理が記述されていることを確認する．

(d) 離散値は定数値（例えば，0, 1）で記述するのではなく，その意味を表す名標（例えば，WIDTH, HIGHT）で定義してから使用する．

(e) プロジェクトの規約により，1 文字違いの変数名は作らないことにする．

(f) 繰り返し処理については，その処理の中に制御変数（繰り返しの終了条件を決定する変数）を更新する処理が記述されていることを確認する．

(k) グローバル変数名はシステムで一意とし，グローバル変数だけからなるソースファイル内にまとめて定義する．

(l) else 節のない if 文については，else 節の処理が必要でないかどうかを確認する．

(m) ポインタ変数の値を参照している箇所では，NULL であるどうかのチェック処理が記述されていることを確認する．

また，留意，工夫をすることにより，バグの数を減らせることができると思われるバグ防止策として，以下のようなものがあった．( ) 内の % は，当該バグのバグ全体 (363 件) に対する割合を示す．

(o) インタフェース解釈誤り (5.8%)

関数仕様機にインタフェース（各パラメータの意味，制約条件，使用例等）をわかりやすく記述する．また，設計／コーディング時に，関数仕様を容易に確認できる設計支援エディタがあるとよい．

(p) 分岐条件記述漏れ (4.4%)

処理対象のデータの取り得る値を常に考慮して，

各々の値に対する処理を設計する．

（q）式の記述における ±1 の誤り（2.8%）
カウンタや座標が0から始まったり，1から始まったりしている．例えば，常に0から始まるように決めておく．

（r）アーギュメントの記述順序の誤り（1.1%）
（o）と同様．

（s）出力すべきメッセージ番号の誤り（0.8%）
設計／コーディング時に，メッセージ内容を確認しながら，メッセージ番号を記述できる設計支援エディタがあるとよい．

（t）エラーメッセージの出力漏れ（0.3%）
関数のリターン値等を判定してエラーを検出した後，エラーメッセージが出力している箇所が存在することを確認できるツールがあるとよい．

5. 分析結果の応用

これまでの分析により，エラーの発生比率，バグ究明時間の分布，バグの存在箇所，バグ発生確率，エラー種別/文種別/バグ種別の関係，それらのバグ究明時間に与える影響，バグの原因，バグ防止策等が明らかになった．ここでは，これらのバグ分析結果の応用方法について考察する．

上記のバグ分析結果をもとに，実際に検討をすすめている2つの応用例について以下に述べる．

5.1 知的デバッグ支援への応用

筆者は，文献 10)において，変数値エラーが発生している場合に，そのエラーを引き起こした可能性のある文を含む最小の集合（Critical Slice）を提案し，この集合を求める方法を明らかにした．この Critical Slice を実行系列（実行された命令の列）上で分割することを繰り返すことにより，バグを含む範囲を段階的に限定し，最終的には，文の記述漏れを含む任意のバグを文単位で検出することを可能としている．

Critical Slice には，代入文，分岐文，ループ文，関数呼出し文等の文が含まれるが，バグ分析結果より，文の種類によってバグ発生確率が異なることがわかっている．例えば，分岐文のバグ発生確率は代入文のバグ発生確率の約3倍である．したがって，Critical Slice をいくつかの部分に分割したとき，各部分のバグを含む確率は，それらの含む文の数と文の種類に依存すると考えられる．そこで，バグ分析結果から得られた文の種別ごとのバグ発生確率を Critical Slice の最適分割アルゴリズムに応用することが考えられる．

5.2 知的レビュー支援への応用

4章で提案したバグ防止策をコーディングレビュー等に適用すれば，バグを早期に発見する点では，ある程度の効果が期待できる．しかし，人間によるチェックには限界がある．そこで，レビューすべき項目をレビュー知識として蓄積し，プログラムレビュー工程においてこれらの知識を用いてプログラムレビュー作業を支援する知的プログラムレビュー支援システム（IRASY）[11]について研究を進めている．レビュー項目は，プログラミングに共通したものばかりではなく，対象とするプログラミング言語/プログラムに依存したもの，プロジェクトに依存した規約，プログラマの経験に依存したもの等，多岐に渡るため，レビュー項目を記述するためのレビュー仕様記述言語をもつ．この IRASY 用の知識ベースとして，バグ分析結果から得られたバグ防止策を蓄えることができた．Russell[12]は，コードインスペクションにより，テスト工程におけるバグ検出工数を大幅に削減できることを示しているが，これを知的に支援することにより，より大きな効果が得られることが期待できる．

6. おわりに

合計 138 KS のソフトウェアの結合試験工程で検出された総計 578 件のバグについて，バグ分析票を基に分析を行い，エラーの発生比率，バグ究明時間の分布，バグの存在箇所，バグ防止策等を明らかにすることができた．主な結果をまとめると，以下のようになる．

（1）エラーの多くは出力異常（62%）という形で発見されている，（2）バグを究明するまでの平均所要時間は40分であり，バグの87%は1時間以内に見つかっている．また，バグの45%は机上デバッグにより検出されている，（3）バグを構文別に分類すると，分岐文のバグが最も多く42%を占める，また，分岐文は1文当たりのバグ発生確率（1.2%）も最も高い，（4）「応答なし」エラーは「文の漏れ」が原因で発生していることが多い，（5）分岐文には文の漏れが多い，（6）エラー種別と文種別との関係は独立である．（7）エラー種別，文種別，バグ種別のいずれもバグ究明時間を決定する要因とはならない，（8）バグ原因の分析に基づいて提案したバグ防止策を実践することにより，約25%のバグを未然に防ぐことができる可能性がある等．

最後に，バグ分析結果の知的デバッグ支援，およ

び，知的レビュー支援への応用について述べた．この他にも，バグの防止策であげたような，誤りをおかさないコーディングを支援してくれるような知的エディタ，あるいは，これらのチェックを自動的に行いプログラマに負担をかけない設計言語／プログラミング言語が開発されることが期待される．

謝辞 本研究を進めるにあたり日頃から励ましと助言をいただきました，NTT ソフトウェア研究所細谷僚一所長，後藤滋樹部長，伊藤正樹リーダに深謝いたします．

参考文献

1) Howden, W. E.: Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Softw. Eng.*, Vol. SE-2, No. 3, pp. 208-215 (1976).
2) Howden, W. E.: Theoretical and Empirical Studies of Program Testing, *IEEE Trans. Softw. Eng.*, Vol. SE-4, pp. 293-298 (1978).
3) Gannon, C.: Error Detection Using Path Testing and Static Analysis, *IEEE Computer*, Vol. 12, No. 8, pp. 26-31 (1979).
4) Sneed, H. M.: Data Coverage Measurement in Program Testing, *Proc. Workshop on Software Teseing* (July 1986).
5) Fitzsimmons, A. and Love, T.: A Review and Evaluation of Software Science, *Computing Surveys*, Vol. 10, No. 1, pp. 3-18 (1978).
6) Nakagawa, Y. and Hanata, S.: An Error Complexity Model for Software Reliability Measurement, *11th International Conference on Software Engineering*, pp. 230-236 (May 1989).
7) Endres, A.: An Analysis of Errors and Their Causes in System Programs, *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 140-149 (1975).
8) Mohri, Y. and Kikuno, T.: Fault Analysis Based on Fault Reporting in JSP Software Develpment, *COMPSAC 91*, pp. 591-596 (1991).
9) 日野克重：ソフトウェア障害の分析と予防，日科技連第4回ソフトウェア生産における品質管理シンポジウム，pp. 90-98 (1984).
10) 下村隆夫：変数値エラーにおける Critical Slice に基づくバグ究明戦略，情報処理学会論文誌，Vol. 33, No. 4, pp. 501-511 (1992).
11) 伊藤智子，下村隆夫：知的プログラムレビュー支援システム（IRASY）の提案，第45回情報処理学会全国大会論文集，5T-9 (Oct. 1992).
12) Russell, G. W.: Experience with Inspection in Ultralarge-Scale Developments, *IEEE Software*, Vol. 8, No. 1, pp. 25-31 (1991).

(平成5年7月 8 日受付)
(平成6年3月17日採録)

下村 隆夫（正会員）

昭和24年生．昭和48年京都大学理学部数学科卒業．昭和 50 年東北大学大学院修士課程修了．同年日本電信電話公社（現 NTT）電気通信研究所勤務．昭和 62 年より NTT ソフトウェア研究所勤務．平成5年12月より ATR 通信システム研究所に出向中．平成4年4月から平成6年3月まで電気通信大学大学院情報システム学研究科客員助教授．これまで，汎用クロスアセンブラ，統計解析プログラム，仕様記述言語，グラフィックパッケージ，CASE ツール，テストカバレッジアナライザ，ビジュアルデバッガ，情報通信システムセキュリティの研究実用化に従事．ソフトウェアの設計，テスト，デバッグの自動化に興味をもつ．電子情報通信学会，ACM 各会員．