プログラムの処理フローに潜むエラーを自動検出
C++test7.0の「バグ探偵」機能

テクマトリックス株式会社 システムエンジニアリング事業部
ソフトウェアエンジニアリング技術部 部長
西田　啓一

C/C++ プログラムの単体テストと静的解析を自動化する『C++test 』にプログラムの処理フローを静的に解析し、エラーを検出する『バグ探偵』機能が追加されました。ここでは、『バグ探偵』について、詳しくご紹介します。

フロー解析とは

まず、C++testの「バグ探偵」機能のエラー検出の基本であるフロー解析についてご説明しましょう。

フロー解析というと、ネットワークのフロー解析などが有名ですが、C++testのフロー解析は、ソースコードを解析して、実行される可能性があるパスをすべて列挙してエラーの有無を解析することを指します。コードの網羅率（カバレッジ）でいえば、パスカバレッジに相当するパスを網羅してエラーを検出します。

具体的な例として図１のフローチャートを参照してください。

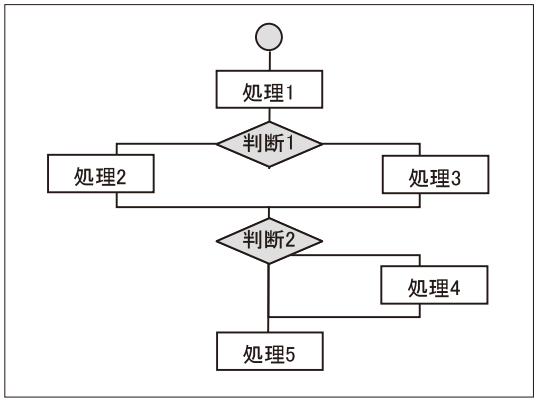

図１　処理フローの例

図１のフローチャートには、図２に示す４つのパスが存在します。C++testの「バグ探偵」は、図２の４つのパスをすべて解析して、フローひとつひとつの中で発生する可能性があるエラーを検出します。

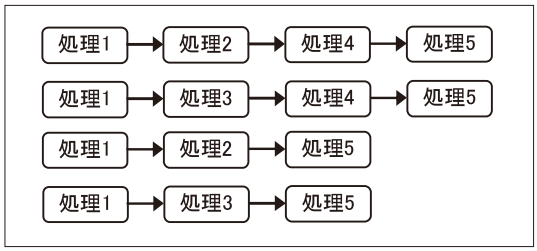

図２　図１の処理フローに含まれるパス

C++testの「バグ探偵」で検出可能なエラーのカテゴリは以下の６つです。

- メモリリーク
- リソースリーク
- NULLポインタを参照する可能性があるポインタ処理
- 未初期化変数を参照する可能性がある処理
- ゼロ除算の可能性がある処理
- 配列の範囲外のアクセス

上記の各カテゴリは、さらにいくつかの細分化されたエラーパターンに分類され、全部で16パターンのエラーを検出します。

では、具体的なエラー検出パターン例をご紹介しましょう。

リソースリーク

リソースリーク（メモリを含む）は、言語によらずプログラマの頭を悩ます大きな問題のひとつです。このエラーは、往々にしてプロジェクトの後半になってから顕在化する傾向があります。開発量が小規模である場合には、比較的

容易に原因を把握できますが、大規模なシステムの場合には、原因の特定が非常に難しくなる問題のひとつです。この問題を、開発プロセスの早期の段階で見つけることができれば、大きなメリットが得られることを、この手の問題に悩まされたプログラマなら、すぐにご理解いただけるはずです。

では、C++testの「バグ探偵」がリソースリークをどのように検出するか、ご覧いただきましょう。

図3の例では、9行目でmallocされた、ポインタ「Person* p 」が、フリーされていません。このようなエラーの場合、以下のようにエラーが発生するまでの詳細なトレース情報がレポートされます。

- **5行で、リークの対象となる変数が宣言されている**
- **9行目で、領域確保されている（mallocの呼び出し）**
- **10、11行目で「Person* p 」の操作を実行**
- **17行目でリーク発生をレポート**

```
01:     void printPersonInfo(FILE* file)
02:     {
03:        char *c;
04:        char buf [100];
05         Person* p;
06:        while(1) {
07:           c = fgets(buf, 100, file);
08:           if(c!=NULL) {
09:              p = malloc(sizeof(Person));
10:              p->personalId = ++id;
11:              p->name = buf;
12:              storePerson(p);
13:           } else {
14:              Break;
15::          }
16:        }
17:     }
```

図3　リソースリークを発生させるソースコード

リークのエラーを修正する際には、上記の情報は非常に重要な意味を持ちます。変数がどこで宣言され、どこで初期化され、どこで参照されているのかが詳細に理解できないと、開放済みポインタの参照などの新たなバグをソースコードに入れ込んでしまう可能性があります。特に、複数の関数やソースにまたがってエラーが発生する場合には、これらの情報は、問題点解決のために必須の情報になります。図4が、C++testのリソースリーク検出の様子です。

図4　C++testのバグ探偵がリソースリークを検出した例

未初期化変数の参照

未初期化変数の参照もリークの問題同様に、厄介な問題のひとつです。例えば、常に一定の値を出力するべき計算結果が、実行のたびに微妙に変化する場合などがこれに相当します。

これについても具体的にその検出例をご覧いただきましょう。

```
01:  void writePersonToFile(Person *person, char* filename)
02:  {
03:      FILE *file;
04:      int numberOfCharactersPrinted;
05:      file = fopen(filename, "w");
06:      if (file!=NULL) {
07:          numberOfCharactersPrinted
08:              = fprintf(file, "Id: %d Name: %s
     References:%s¥n",
09:          person->personalId, person->name,
     person->reference);
10           fclose(file);
11:       }
12:       printf("File length: %d¥n",
     numberOfCharactersPrinted);
13   }
```

図5　未初期化変数の参照エラーを発生させるソースコード

図5では、変数numberOfCharactersPrintedは、fileがNULLの場合、初期化されずに12行目で参照されます。この問題を的確に修正するための以下の情報を表示します。

・未初期化変数の宣言箇所
・最初の参照箇所

ゼロ除算

エラー検出例の最後にゼロ除算エラーの検出例をご覧いただきましょう。

図6では、27行目の戻り値が0なので、processStaff呼び出しの第2引数が0になり、その結果、calculateAverageSalaryで実行されている割り算「WAGE_FUND / numberOfEmployees;」がゼロ除算のエラーとして報告されます。

```
01:  int calculateAverageSalary(int numberOfEmployees)
02:  {
03:      int WAGE_FUND = 10000;
04       return WAGE_FUND/numberOfEmployees;
05:  }
06:  void processStaff(Person* employees[], int sizeOfStaff)
07:  {
08:      int i;
09:      for (i = 0; i < sizeOfStaff; ++i) {
10:          displayPersonalInfo(employees[i]);

11:      }
12:      printf("Average salary: %d¥n",
                 calculateAverageSalary(sizeOfStaff));
13:  }
14:  int processFile(FILE* file, Person* employees[])
15:  {
16:      return 0;
17:  }
18:  int main()
19:  {
20:      FILE *file = fopen("staff.txt", "r");
21:      Person* employees[100];
22:      int numberOfEmployees;
23:      if(file==NULL) {
24::         printf("Error: file can not be opened.¥n");
25:          return 1;
26:       }
27:       numberOfEmployees = processFile(file,
              employees);
28:       processStaff(employees, numberOfEmployees);
29:       fclose(file);
30:       return 0;
31:  }
```

図6　ゼロ除算を発生させるソースコード

図7では、C++testの「バグ探偵」がフロー解析に基づいて、

Main → processFile → main → processStaff
→ calculateAverageSalary

の流れでエラーが発生することをレポートしています。

図7　C++testのバグ探偵がゼロ除算を検出した例

「バグ探偵」の利点

C++testの「バグ探偵」が採用するエラー検出方法は、実際にプログラムを動作させてエラーを見つける方法にくらべて、以下の点で優れています。

・検証用のコードを準備する必要がない
・エラーのとり漏れがなくなる

プログラムを動作させて実施するエラー検出は、テストを行う際のテストケースに大きく依存します。例えば、図1で示したフローの場合、最後の処理を通るテストケースの実行が漏れた場合、このフローで発生するエラーは検出できません。しかし、C++test「バグ探偵」のエラー検出方法では、検証の網羅率が非常に高く、エラーの検出漏れはなくなります。

また、「バグ探偵」は、プログラムを動作させていないので、開発の早い段階から検証を実施することができます。開発後期で発見されたエラーは、そのデバッグ作業に数日を費やしてしまうような困難なものもあります。コーディング工程で「バグ探偵」を使用してエラーの検出し、デバッグすることにより、開発後期のテストで発見される重大なバグを未然に防ぎ、負担の大きいデバッグ作業を軽減させることができるのです。

「バグ探偵」の運用

C++testの「バグ探偵」は、ソフトウェアの

品質向上において非常に効果的であり、人的負荷の少ないアプローチでもあります。しかし、「バグ探偵」はソースコード中のすべてのパスを解析するので、検証実行中に多くのマシンリソースを使用する場合があります。従って、夜間などの比較的マシンリソースに余裕がある時間に実施すると効率よく「バグ探偵」を活用することができます。

図8はC++test の「バグ探偵」の運用例です。

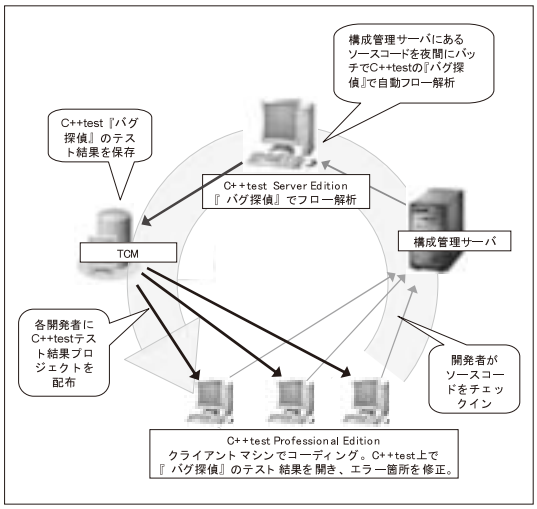

図8　C++test の「バグ探偵」の運用例

図8では、開発者がチェックインしたソースコードに対して、夜間に「バグ探偵」による検証をバッチで自動実行し、その結果をTCM（Team Configuration Manager）を経由して各開発者にフィードバックします。この運用例の場合、開発者は自身のマシンで「バグ探偵」による検証を実施する必要がないので、その分の時間を削減できるとともに、検証漏れを防ぐことができます。

C++testは、単体テストやパターンマッチング形式の静的解析を自動化する機能も備わっています。それらもご紹介しましょう。

自動単体テスト

C++testには、テストケースやテストスタブ、テストドライバを自動生成し、単体テストを自動化します。GCCまたはVisualC++でコンパイルできるソースコードであれば、クリックひとつでテストケースの生成から単体テスト実行までを自動的に実施します。人手による単体テストでは、テストスタブやテストケースの作成などの準備に多くの時間を取られてしまいますが、C++test で単体テストを自動化することにより、単体テストの効率化が図れます。

６種類のカバレッジ情報

C++testでは、単体テスト時に、行、ブロック、パス、判断、条件、MC/DC の６種類のカバレッジ情報をレポートします。これらのカバレッジ情報から、単体テストの妥当性を確認することができます。

900種類のコーディングルールで静的解析

C++testには、「Effective C++」や「MISRA」、「C++ Coding Standards」、「Qtベストプラクティス」といったC/C++のコーディングルールが約900種類、搭載されています。これらのコーディングルールでソースコードを検証し、エラーを引き起こす可能性のあるコードや保守性、可読性の低いコードを検出します。また、コーディングルールは、編集したり、ユーザ独自のものを作成したりすることもできるので、プロジェクトや社内の規約をC++testで検証することも可能です。

単体テスト、静的解析、フロー解析で高品質なソフトウェアの開発を支援する、C++testでプログラムの品質向上に挑戦しませんか？

体験版はこちらから	http://www.techmatrix.co.jp/products/quality/ctest/zipc/

【開発元】PARASOFT® We make software work.™
Parasoft Corporation

【総販売代理店】TechMatrix
テクマトリックス（株）　システムエンジニアリング事業部
ソフトウェアエンジニアリング営業部　ソフトウェアエンジニアリング営業課
TEL 03-5792-8606 FAX 03-5792-8706
E-MAIL Parasoft-info@techmatrix.co.jp
URL http://www.techmatrix.co.jp/products/quality/ctest/