ソフトウェアテストシンポジウム 2003
JaSST'03: Japan Symposium on Software Testing 2003

プログラムの実行履歴を用いた利用モデルの作成方法について

高木 智彦† 古川 善吾‡

香川大学 〒761-0396 香川県高松市林町 2217-20

E-mail: †s02g463@stmail.eng.kagawa-u.ac.jp, ‡zengo@eng.kagawa-u.ac.jp

あらまし 統計的テストは，大量のテストケースを用いて，ソフトウェアの実際の利用状況における信頼性を評価できるため，今後有効な手法の一つになると考えられる．ただし，ソフトウェアの利用状況の要約である利用モデルの作成にコストがかかる点が問題である．そこで本稿では，プログラムの実行履歴を取得する GCC(GNU Compiler Collection)およびテストカバレッジツール gcov の機能を利用して効率的に利用モデルを作成する方法を考察する．本手法は，ソースコードと状態遷移図を対応付けるための標識コメントの挿入に手間がかかるものの，過去のバージョンが利用可能であり，また，テスト工程で新たに図を用意する必要がないなど，有用な点が認められた．

キーワード 統計的テスト，利用モデル，状態遷移図，実行履歴，GCC

Constructing Method for Usage Model using Execution History of Program

Tomohiko TAKAGI† Zengo FURUKAWA‡

Kagawa University 2217-20 Hayashi-cho, Takamatsu City, Kagawa 761-0396, JAPAN

E-mail: †s02g463@stmail.eng.kagawa-u.ac.jp, ‡zengo@eng.kagawa-u.ac.jp

Abstract Statistical testing, a method using a lot of testcases and evaluating software's reliability on actual usage, has a problem that usage model is constructed at high cost. So this paper proposes a constructing method for usage model with program's execution history using GCC and gcov. There is some useful points in this method, e.g. a past version of software is applicable and we don't have to describe a new diagram in testing.

Keyword Statistical Testing, Usage Model, State Transition Diagram, Execution History, GCC

1. はじめに

近年，ソフトウェアは日常生活のあらゆる場面に浸透したため，バグが社会に及ぼす影響は大きいものとなっている．しかしながら，従来のように，ソフトウェアのすべての機能を網羅的にテストするだけでは，要求される信頼性を十分に満足できない場合がある．

統計的テスト[1][2]は，大量のテストケースを用いて，ソフトウェアの実際の利用状況における信頼性を評価できるため，今後有効な手法の一つになると考えられる．一般的な手順は，(1) ソフトウェアの仕様である状態遷移図に利用状況を関連付けてマルコフ連鎖(すなわち利用モデル)を作成し，(2) 利用モデルの遷移確率を満たすパス(状態と遷移の列)をテストケースとして生成する．(1)では，ソフトウェアのプロトタイプや過去のバージョンの実行履歴などを用いて利用状況を明らかにする．ただし，これまでの研究において，実行履歴を状態遷移図に関連付ける具体的方法は，ほとんど論じられていない．さらに，利用モデルの作成にコストがかかる点が問題である．そこでプログラムの実行履歴を取得する GCC(GNU Compiler Collection)およびテストカバレッジツール gcov[3]を利用して効率的に利用モデルを作成する方法を考察する．

本稿では，まず 2 節で統計的テストについて概観する．そして 3 節で利用モデルの作成方法について，4 節でサンプルプログラムへの適用例を説明する．最後に，5 節でテストケース作成支援システムを紹介し，6 節で本手法に対する考察を行う．

2. 統計的テスト

2.1. 概要

統計的テストは，クリーンルーム開発手法におけるテスト法として開発された．クリーンルーム開発手法によって作成されたコードはプログラミング終了時点での品質が高いため，バグを発見するためではなく，

品質を測定するためにテストを行う．ただし本研究では，クリーンルーム開発手法によって作成されたコードのみを対象にしているのではない．むしろ，従来の開発手法によって作成されたコードへの適用を目的としている．

ソフトウェアの品質とは，ユーザにとっての外部的性質である．例えば，ソースコード 1000 行当たりのバグの数は，ユーザの視点からは全く意味がない．つまり，コード中にバグを含んでいても，1 回も実行されることがないのであれば問題はない．逆に，ユーザがよく使う機能を記述しているコードにはバグを含んではならない．

そこで統計的テストでは，ユーザの使用特性を外部仕様の観点から統計的に分析し，テストケースに反映させる．使用頻度の高い機能にテストの重心を置くことによって，ユーザがよく使う機能に含まれるバグを重点的に検出することが期待できる．

2.2. 従来の手法との比較

統計的テストは，ユーザの立場から行い，ソフトウェアそのものの信頼性を求める．テスト十分性は MTTF (Mean Time To Failure)の測定結果による．MTTF とは，ある欠陥が発生してから，次の欠陥が発生するまでの平均時間である．出荷基準に達したかどうかは，MTTF がある一定以上になったかどうかで判断する．統計的テストでは，原理的に出荷した後の使用環境において，障害発生確率の高いものが欠陥として起こる．ゆえに，効果的なテストを行うために，利用モデルがユーザの使用特性を反映していることを要求する．

一方，従来のテストは，開発者の立場から行う．すなわち，プログラムや仕様に基づいて網羅的にテストを行い，作業の品質(全体のうちのどれだけを網羅したか)によって対象となるソフトウェアの信頼性を予測している．この手法は，コード上のすべてのパス(経路)を少なくとも 1 回実行することを要求する．しかし，繰り返し処理を含む場合はパスの数が無限に存在するため，すべてのパスを実行できない．そこで，実際にはすべてのパスをテストする代わりにいくつかの簡易化された網羅基準を用いる．網羅基準の種類としては，C_0 (全文網羅)や C_1 (全分岐網羅)などがある．C_0 は，すべてのステートメントを 1 回以上実行するパスセットを選ぶ基準であり，C_1 は，すべての分岐を 1 回以上実行するパスセットを選ぶ基準である．テスト十分性は網羅基準を満たしたかどうかで判断する．

2.3. テスト手順

これまでに提案されている統計的テストの一般的な手順を以下に示す．

1. 利用モデルの作成
   ソフトウェアの仕様として作成される状態遷移図，およびユーザのソフトウェアに対する使用特性を用いてマルコフ連鎖を作成する．マルコフ連鎖とは，状態遷移図において遷移先が確率的に決定されるものをいう．これが利用モデルである．利用モデルでは，ソフトウェアに対する，ユーザ，利用の仕方，環境などを定義できる．利用の仕方は，ユーザの動作によって定義される．環境は，ソフトウェアが動作するプラットフォームや外部データベースなどの考慮すべき要因である．利用の状況についても，夜間と昼間，緊急時と平常時などに分類する．
2. テストケースの作成
   利用モデルから，各遷移の遷移確率に従い，初期状態から終了状態に至る状態と遷移の列としてパスを作成する．作成したパスは，ソフトウェアに対する入力条件であり，かつ期待出力でもあるため，テストケースと見なすことができる．
3. テストの実施およびテストモデルの作成
   テストケースを実行し，得られた出力を状態遷移図に記入する．誤りが発生した時には，誤り状態を新たに追加し，その誤り状態への遷移を状態遷移図に追加する．誤り状態からの遷移は，致命的な誤り(回復できない誤り)ならば終了状態への遷移を，また，軽微な誤り(回復できる誤り)ならば元の状態への遷移を追加する．これがテストモデルであり，テストの履歴として利用する．
4. 信頼性評価
   誤り状態に達しない確率を求めることによって，信頼性を評価する．また，誤り状態に達するまでの実行回数から，MTTF を求めることができる．
5. テスト終了判定
   以下のような指標に基づいてテストの十分性を判定する．
   - 利用モデルとテストモデルの差が与えられた閾値以下．
   - 信頼性が与えられた目標値より大きい．
   - MTTF が与えられた目標値より大きい．

テスト中にバグが発見された場合は，発見毎にプログラムの修正は行わずに，テストを一通り終わらせることを優先する．修正後において，どのレベルのテストからやり直すかは場合による．その際，データフロー解析に代表されるプログラム・スライシング技術によって影響範囲を抽出することが多い．

3. 利用モデルの作成

3.1. 作成手順

本研究で考察した，プログラムの実行履歴による利用モデルの作成手順について，以下にまとめる．

1. 状態遷移図とプログラムを用意する
   ソフトウェアの仕様である状態遷移図を作成する．状態遷移図は，ユーザにとって意味のあるレベルまで詳細化する．また，最終的にテストケース生成に用いるため，入出力に関する機能を漏れなく記述する必要がある．プログラムについては，C 言語を前提とする．
2. プログラム中に標識コメントを挿入する
   状態遷移図の状態や遷移に対応するプログラムのコード行に，標識コメント(以降では単に「標識」と呼ぶ)を挿入する．標識は，状態名や遷移名で表す．これによって，特定のコード行が状態遷移図のどの状態や遷移に対応しているかを識別できる．
3. プログラムを実行する
   GCC を用いてプログラムをコンパイルし，実行する．GCC に付属のテストカバレッジツール gcov を利用することによって，プログラムの各コード行が実行される頻度(累積の実行回数)を知ることができる．
4. 実行履歴を状態遷移図に関連付ける
   標識を手がかりにして，コード行の実行回数を状態や遷移に関連付ける．そして遷移確率を計算し，利用モデルを完成させる．実行回数を関連付けることができなかった状態や遷移(プロトタイピングの対象外の機能や，新たに追加する機能などに相当する)について，一定の遷移確率を別途割り振る必要がある．

3.2. 標識の挿入位置

本手法では，標識の直後のステートメントの実行回数を，状態や遷移の実行回数とみなしている．従って，正確な実行回数を取得するためには，標識をソースコードの適切な位置に挿入しなければならない．特に手動で行う場合は，作業の効率や信頼性の点から，挿入するべき位置を明確に決めておくことが重要である．

そこで，以下に挿入位置の候補を列挙する．項目番号は優先順位を表している．この優先順位に従って，必要なだけ標識を挿入する．

1. 関数宣言の直前(状態に対応)
   状態は 1 つのまとまった機能を構成しているため，関数に対応付け可能な場合が多い．ただし，関数が複数の状態に対応している場合は，挿入位置として不適当である．
2. 関数呼び出しの直前(遷移に対応)
   関数が状態に対応している場合，その関数の呼び出しは状態への遷移と見なすことができる．ただし，繰り返し構文(for 文や while 文など)の条件部で関数呼び出しを行っている場合は，繰り返しの回数がカウントされない場合があるので注意を要する．
3. 入出力ステートメントの直前(状態に対応)
   関数 scanf などによる，ユーザからの入力待ちを状態と見なすことができる．
4. その他
   状態に対応する関数が存在せず，状態内に入出力ステートメントが存在しなければ，上記 1,2,3 を適用できない．この場合は，作業者の判断で挿入位置を決定する．あるいは，状態に対応するソースコードを関数化すれば，1,2 が適用できる．
   ラムダ遷移については，作業者の判断で，実行回数の点で等価なステートメントに関連付ける．なお，ラムダ遷移とは，現在の状態に対応する処理が終了した時点で，次の状態に遷移を行うものをいう．一般的に，ラムダ遷移にラベルは設定されない．

ソースコードが以下の条件を満たしていれば，手順に従って比較的容易に標識を挿入できる．

- 状態に対応するソースコードが関数化されており，かつ，関数が複数の状態に対応していない，または，
- 状態に対応するソースコード中に入出力ステートメントを含んでいる．

3.3. 標識挿入アルゴリズム

利用モデルを作成するには，少なくともすべての遷移の実行回数を明らかにする必要がある．そこで，すべての遷移をコード行に対応付けるための標識挿入アルゴリズムの一例を以下に示す．

1. 状態と関数を対応付ける．
2. 遷移と関数呼び出しを対応付ける．
3. 状態と入出力ステートメントを対応付ける．
4. 入力(出力)遷移を 1 本しか持たない状態について，もし状態が関数 F に対応していれば，入力(出力)遷移を F に対応付ける．同様に，もし状態が入出力ステートメント S に対応していれば，入

力(出力)遷移を S に対応付ける.

5. 残りの標識を作業者の判断で挿入する.

1.は，状態に対応する関数のスケルトンコードと標識を自動生成することによって実現できる．2.においては標識の挿入を自動化するとよい．4.はラムダ遷移を考慮したもので，これも自動化が可能である．

対応付けられた状態や遷移についてはフラグを立てておき，以降の作業対象から除外する．すべての遷移のフラグが立った時点で挿入作業は終了する．式(1)から実行回数を算出できる遷移については，無理に対応付けなくてもよい．

(状態の実行回数)
＝(すべての入力遷移の実行回数) ･･･ (1)
＝(すべての出力遷移の実行回数)

4. 適用例

3 節では，標識を用いて利用モデルを作成する方法を説明した．そこで，この節では簡単なアドレス帳プログラムに対する実際の適用例を示す.

4.1. アドレス帳プログラムの仕様

図 1 に,アドレス帳プログラムの状態遷移図を示す.

主な機能は,「データ追加」,「データ一覧」および「データ削除」である.「データ追加」では，氏名，住所，電話番号，備考を 1 レコードとして，アドレスデータを追加登録する.「データ一覧」では，登録済みのすべてのアドレスデータを一覧表示する.「データ削除」では，氏名をキーとして検索を行い，該当するレコードを削除する.

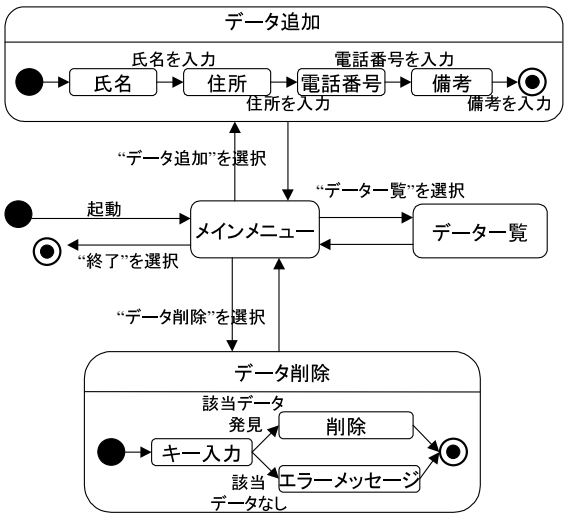

図 1： アドレス帳プログラムの状態遷移図

4.2. アドレス帳プログラムの利用モデル作成

図 1 に基づいて作成されたソースコードに，標識を挿入する．図 2 は，図 1 中の状態「メインメニュー」に対応する関数である.

```
/* std:メインメニュー */
int menu_screen(void) {
  int num;
  do {
    printf("*****************************¥n");
    printf("*  アドレス帳メインメニュー  *¥n");
    printf("*****************************¥n");
    printf("〇 機能を選択して下さい. ¥n");
    printf("データ追加 : 1   データ削除 : 2¥n");
    printf("データ一覧 : 3   終了       : 4¥n");
    scanf("%d", &num);
  } while (num < 1 || num > 4);
  return num;
}
```

図 2： 関数と状態の対応付け

関数と状態が 1 対 1 に対応する場合は，図 2 のように関数宣言の直前に標識を挿入すればよい.

メインメニューから各状態への遷移については，図 3 のように標識を挿入する.

```
while ((menu_item = menu_screen()) != END) {
  switch (menu_item) {
    case ADD:
      /* std: "データ追加"を選択 */
      add_item();
      break;
    case DELETE:
      /* std: "データ削除"を選択 */
      delete_item();
      break;
    case DISPLAY:
      /* std: "データ一覧"を選択 */
      display();
      break;
  }
}
```

図 3： 関数呼び出しと遷移の対応付け

以上のようにして標識を挿入した後，プログラムを GCC でコンパイルする．その際，-fprofile-arcs と -ftest-coverage のオプションを指定することによって，gcov が必要とするプロファイル情報を生成するコードを追加できる．プログラムを実行後，gcov を用いて各コード行の実行回数を取得すると図 4 や図 5 のようになる．各ステートメントの左端の数字は，そのステートメントの実行回数を意味する．このような実行履歴情報から，標識を手がかりに利用モデルを作成したものが図 6 である．すべての遷移に遷移確率と実行回数を付している.

```
     /* std:メインメニュー */
     int menu_screen(void) {
332    int num;
332    do {
332      printf("*****************************¥n");
332      printf("*  アドレス帳メインメニュー  *¥n");
332      printf("*****************************¥n");
332      printf("○ 機能を選択して下さい. ¥n");
332      printf("データ追加 : 1   データ削除 : 2¥n");
332      printf("データ一覧 : 3   終了       : 4¥n");
332      scanf("%d", &num);
332    } while (num < 1 || num > 4);
332    return num;
332  }
```

図 4： 図 2 のコードの実行履歴例

```
332  while ((menu_item = menu_screen()) != END) {
205    switch (menu_item) {
 60      case ADD:
          /* std: "データ追加"を選択 */
 60         add_item();
 60         break;
         case DELETE:
          /* std: "データ削除"を選択 */
  9         delete_item();
  9         break;
         case DISPLAY:
          /* std: "データ一覧"を選択 */
136         display();
136         break;
205    }
205  }
```

図 5： 図 3 のコードの実行履歴例

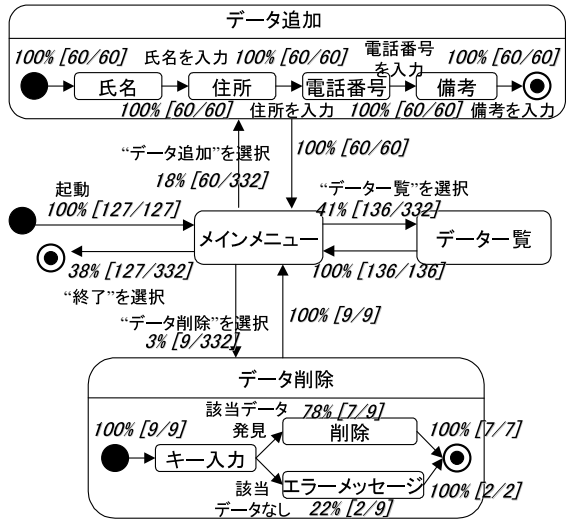

図 6： アドレス帳プログラムの利用モデル

4.3. 追加の状態を含む利用モデルの作成

このプログラムの次のバージョンでは，新たに「検索データ表示」の機能を追加した(図 7 の網掛け部分)．「検索データ表示」は，登録済みのすべてのアドレスデータの中から，ユーザが入力した氏名をキーとして検索を行い，該当するレコードを表示する．この新しい機能(状態)を含むプログラムの利用モデル作成は以下のようになる．

- 前バージョンの状態遷移図(図 1)から変更のない部分には，前バージョンの利用モデル作成に用いた実行履歴(すなわち，図 6 中の遷移確率と実行回数)を適用する．
- 新しく追加したサブ状態遷移図(図 7 の状態「検索データ表示」の中に含まれる図)については，遷移確率を一様に設定する．
- 既存の状態に新たに追加した出力遷移(図 7 の「"検索データ表示"を選択」)には，遷移確率として平均確率（100/[遷移先数]．ある状態からのすべての遷移が等確率で発生したとしたときの確率.）を設定する．あるいは，作業者の判断で確率を設定してもよい．既存の遷移の遷移確率は，同一割合で減少させる．

以上に従って作成した利用モデルが図 8 である．

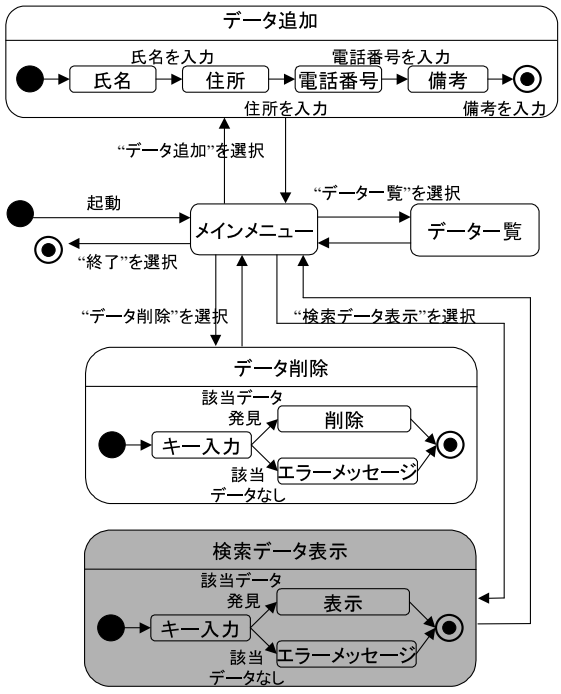

図 7： 新機能を追加したアドレス帳の状態遷移図

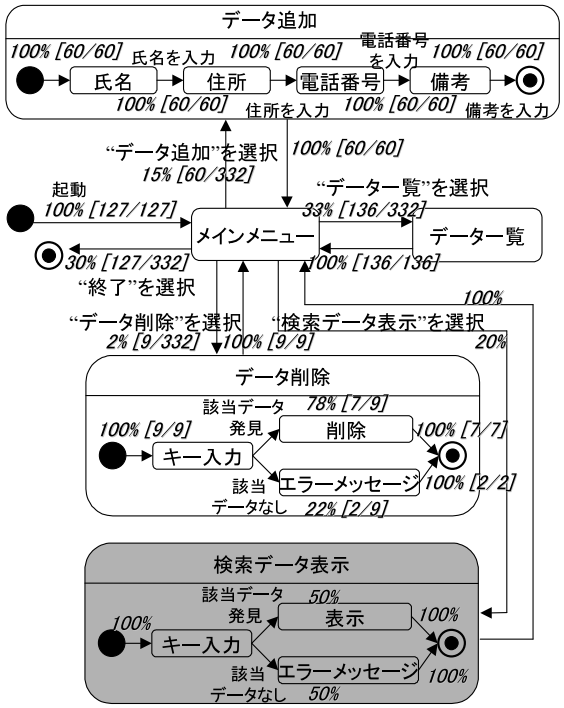

図 8： 新機能を追加したアドレス帳の利用モデル

4.4. テストケース生成

利用モデル(図 8)からアドレス帳プログラムのためのテストケースを生成することができる．テストケースは，利用モデルの遷移確率を満たすパス(初期状態で始まり終了状態で終わる，状態と遷移の列)で表される．以下は，テストケースの例である．テストケース中の「状態名（...)」は，サブ状態のパスを示している．

- 「初期状態」→起動→「メインメニュー」→“データ追加”を選択→「データ追加」（「初期状態」→ラムダ遷移→「氏名」→氏名を入力→「住所」→住所を入力→「電話番号」→電話番号を入力→「備考」→備考を入力→「終了状態」）→ラムダ遷移→「メインメニュー」→“データ一覧”を選択→「データ一覧」→ラムダ遷移→「メインメニュー」→“終了”を選択→「終了状態」

テストケース中のユーザ入力に関する遷移を抽出すると，テスト入力シーケンスを得ることができる．

- 起動→“データ追加”を選択→（氏名を入力→住所を入力→電話番号を入力→備考を入力）→“データ一覧”を選択→“終了”を選択

最終的には，テスト入力シーケンスに対して同値分割法や限界値分析法などを適用し，テストデータを導出する．

5. テストケース作成支援システム

この節では，参考文献[2]で開発したテストケース作成支援システムの概要を述べる．そして，本稿で提案した手法をシステムにどう実装すべきかを考える．

5.1. 概要

統計的テストでは，利用モデルを作成したり，信頼性の評価に十分足りる量のテストケースを作成したりする必要があるため，従来のテスト方法より時間がかかる．そこで参考文献[2]において，作業を自動的に行うために,テストケース作成支援システムを試作した．システムは主に，Graph Editor，Markov Modeler，Path Analyzer の 3 つのモジュールから構成される(図 9)．このシステムを用いることによって，利用モデルから大量のテストケースを自動生成することが可能である．

- Graph Editor (図 10)
  GUI を用いた状態遷移図の編集や,利用モデルの表示，状態遷移図のエラーチェックなどの機能を提供する．システムの中心的モジュールであり，ここから他のモジュールを呼び出す．

- Markov Modeler (図 11)
  統計的テストのためのテストケースとして状態遷移図からパスセットを自動作成する．手順を以下に示す．
  (1) Graph Editor で編集した状態遷移図上の状態をクリックして実行系列(ソフトウェアに対するユーザの使用特性)を入力する．
  (2) 入力した実行系列を用いて，利用モデルを作成する．
  (3) 利用モデルに従い，各遷移の遷移確率を満たすパスをテストケースとして自動生成する．

- Path Analyzer (図 12)
  ユーザが指定した始点と終点および網羅基準に基づくパスとして，状態遷移図からテストケースを自動生成する．網羅基準は，全状態網羅(すべての状態を 1 回以上実行するパスセットを選ぶ基準)と全遷移網羅(すべての遷移を 1 回以上実行するパスセットを選ぶ基準)がある．統計的テストでは，すべての機能を網羅することが保証されないので，このモジュールを用いてテストケースを補完する．

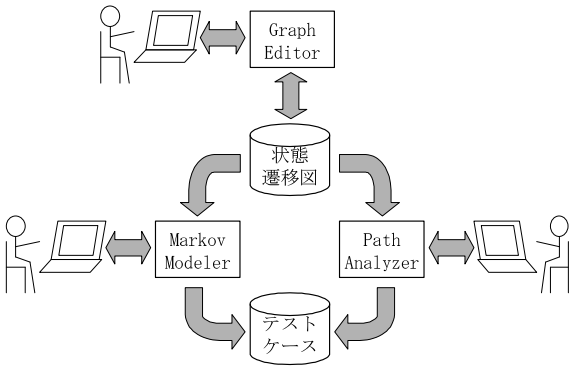

図 9： システム構成

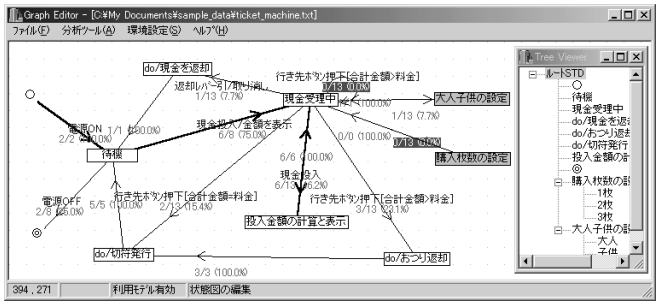

図 10： Graph Editor 実行画面

図 11： Markov Modeler 実行画面

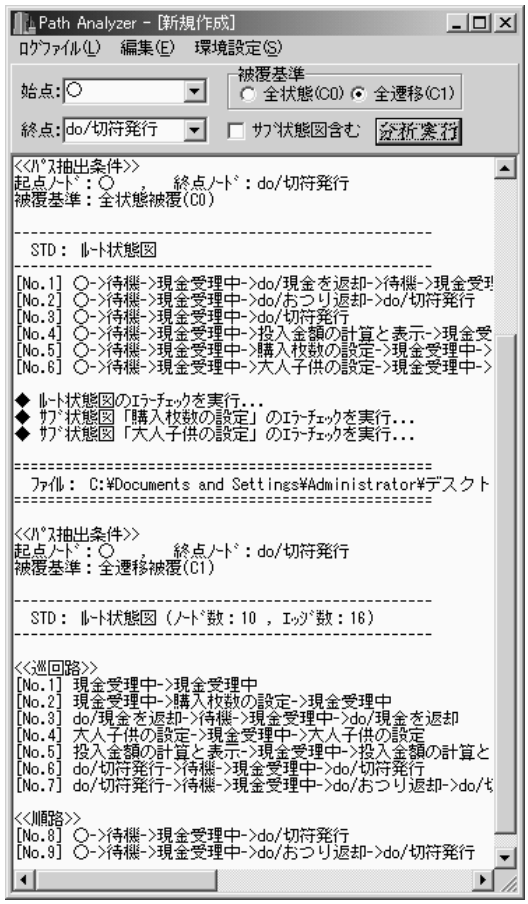

図 12： Path Analyzer 実行画面

5.2. 本手法のシステムへの実装

本手法は現在，システムに実装されていない．システムによって本手法を自動化するには，少なくとも以下に示す機能を追加する必要がある．

- 状態に対応する関数のスケルトンコードと標識の自動生成機能
- 挿入すべき標識のリストや挿入位置の候補をユーザに示すためのガイド機能
- ソースコードと状態遷移図の間の，標識を介した相互参照機能
- プログラムの実行履歴ファイルを分析する機能

6. 考察

本手法の意義を以下にまとめる．

- 図の流用が可能
  テスト工程で，新たに図を作成する必要がない．仕様化工程で作成した状態遷移図をそのまま用いることができる．
- 過去のバージョンが利用可能

状態遷移図の要素に対応してさえいれば，プログラム構造が異なる過去のバージョンやプロトタイプでも適用できる．

- 仕様とプログラム間の矛盾点の発見
  プログラムを状態遷移図に対応付ける作業を通して，プログラムと状態遷移図の間の矛盾点を発見できる可能性がある．
- ユーザインタフェースの改善
  利用モデル上で，使用頻度の高い機能がある場合は，ショートカットキーを作成したり，操作を簡素化することによって，ユーザインタフェースを改善できる可能性がある．逆に，使用頻度が極端に低い機能がある場合は，その機能の必要性が低いか，あるいは，ユーザインタフェースに問題があってユーザがその機能にたどり着くことができない可能性を疑うことができる．

一方，本手法の問題点は以下の通りである．

- 標識挿入の煩雑さ
  実行回数を正確に取得するには，標識を正確な位置に挿入する必要がある．対話型システムや 3.2 節の条件を満たしているものを除き，挿入作業が煩雑になりやすい．
- テストモデルの作成が不可能
  プログラムの実行中に致命的なエラーが発生して強制終了した場合，gcov はそのセッションの実行履歴を取得できない．したがって，テストモデルの構築が不可能である．

7. おわりに

本稿では，プログラムの実行履歴を取得する GCC および gcov の機能を利用して，効率的に利用モデルを作成する方法を考察した．本手法は，ソースコードと状態遷移図を対応付けるための標識の挿入に手間がかかるものの，過去のバージョンが利用可能であり，また，テスト工程で新たに図を用意する必要がないなど，有用な点が認められた．

今後の研究においては，テストの実施や信頼性評価などについて考察する．また，テスト支援システムの試作を進めて，本手法の客観的な有効性を調査する予定である．

文　献

[1] J. A. Whittaker and M. G. Thomason : "A Markov Chain Model for Statistical Software Testing", IEEE Transactions on Software Engineering, October 1994, pp.812-824.

[2] 高木智彦，古川善吾："UML 状態図を用いたテストケース作成支援システムの試作"，情報処理学会研究報告，Vol.2002，No.23，pp.79-86.

[3] Richard M. Stallman : "Using and Porting the GNU Compiler Collection (GCC) "， Free Software Foundation，1999