ソフトウェア工学 119－6
（1998．5．15）

# プログラムの構造テスト支援

廣田豊彦† 橋本正明†

プログラムが高度化したり，また再利用が促進されるようになると，プログラムのテストはより一層重要になり，より高度なテストが要求されるようになってくる．そのような場合でも，高度なテストの基礎として，全文テストや全分岐テストなどの構造テストを確実に行うことをおろそかにすべきではない．しかし，全文テストや全分岐テストを 100% 実施することは必ずしも容易ではない．著者らは，あるプログラムコンポーネントの開発において，テストドライバとスタブを工夫することによって，100% 網羅を達成した．本報告では，そのテストドライバとスタブを分析した結果について述べ，それらの自動生成について考察する．

# Support for Structural Testing of Programs

TOYOHIKO HIROTA† and MASAAKI HASHIMOTO†

As programs become complex and program reuse is encouraged, program testing becomes more important and higher level of testing is required. Even in such cases, as a basis of higher level of testing, all-statement or all-branch testing should be certainly performed. However it is not an easy task to cover all-statement or all-branch testing to 100%. The authors achieved 100% coverage in a certain program component development by preparing elaborate test drivers and stubs. This report describes the analysis of these test drivers and stubs and discuss about automatic generation of them.

## 1. はじめに

ソフトウェアの信頼性を高めるために様々な工夫がなされるが，その最も基本となるのがテストである．無計画にデバッグの手助けとして実施されるようなテストはあまり信頼性の向上には貢献しないが，事前にテストを設計してプログラムを開発することは，テストを実施すること以上にソフトウェアの信頼性に貢献するといわれている[1]．

プログラムのテストは大きく機能テストと構造テストとに2分される．機能テストは対象プログラムをブラックボックスとみなして，ある入力を与えたときに，仕様通りの出力が得られるかどうかを検証する．それに対して構造テストは，対象プログラムの内部構造に基づいて設計され，実施される．

構造テストの代表的なものは，1) 全文テスト，2) 全分岐テスト，3) 全パステストの三つである．すべての文を実行するようにテストケースを設計することは容易ではないが，実際にテストが実行されたときに，すべての文が実行されたかどうかを測定することは容易である．Unix の tcov コマンド[2] を使うと，C または Fortran で書かれたプログラムに対して，全文テストを実行したかどうかを確認することができる．全分岐テストは全文テストを包含し，さらに新たな文を含まない分岐のテストを要求する．たとえば *else* のない *if* 文に対しては，全文テストでは条件が偽となる場合をテストする必要はないが，全分岐テストでは条件の真と偽がともにテストされる．

一般に，構造テスト（おもに全文テストや全分岐テスト）だけでは，プログラムのテストとしては不十分であり，プログラムの性質などに応じて，様々なテストを組み合わせる必要がある[3]．しかし，構造テストが不要であるというわけではなく，あらゆるテストの基礎として，確実に達成する必要がある．ところが，全文テストや全分岐テストを 100% 実施することは，さまざまな理由から，それほど容易なことではない．本報告では，あるプログラムに関して，全文テストを 100% 実施した例について分析し，テストドライバ，スタブの自動生成について考察する．

## 2. テスト支援環境

テスト支援環境は，以下のような機能を持つ必要がある．

(1) ドライバ，スタブの提供
(2) オブジェクト指向への対応

† 九州工業大学情報工学部知能情報工学科
Department of Artificial Intelligence, Kyushu Institute of Technology

(3) テストの自動的な実行

これらの要求について，具体的な背景を述べた後，それらに対応するテスト実行プログラムの構成について述べる．

### 2.1 ドライバ，スタブの提供

一般にプログラムのテストは，システムテストなどのような完結したプログラムを対象とする場合を除いて，適切なテストドライバを準備して，テスト対象コンポーネント（プログラムモジュールや，モジュールの集まり）を呼び出す必要がある．さらに，テスト対象コンポーネントが他のコンポーネントを呼び出している場合，それらのコンポーネントを準備する必要がある．そのような呼び出されるコンポーネントがすでに存在していて，利用可能であれば，それを利用すればよいが，そうでないときには，いわゆるスタブを準備する必要がある．実は，テスト対象コンポーネントを徹底的にテストするためには，たとえ呼び出しているコンポーネントが実在していても，スタブを準備する必要があるし，同様に，システムコールやライブラリ呼び出しに対してもスタブを必要とする．というのは，テスト対象コンポーネントは，将来別の環境で再利用される可能性があるからである．ソフトウェア開発の効率化のためには，ソフトウェアの再利用は不可欠であり，その意味から，プログラムのテストにおいても，ソフトウェアの再利用を念頭におかなければならない．

### 2.2 オブジェクト指向への対応

近年はオブジェクト指向によるソフトウェア開発が一般化してきている．オブジェクト指向プログラムの場合，テストの最小単位はオブジェクトである．従来の手続き型プログラムでは手続きや関数が最小単位であって，それらは原則として1入口／1出口である．それに対して，一つのオブジェクトには一般に複数のメソッドがある．各メソッドを一つの手続きや関数とみなしてテストするだけではオブジェクトのテストとしては不十分である．また，メソッドの集合を，コンポーネントの集合と同列に扱うことも適切でない．というのは，コンポーネント間は明示的なインタフェース（パラメータ）を介して緩やかに結合されるのに対して，同一オブジェクト（クラス）内のメソッドは，共有の属性を介して緊密に結合されているからである．そこで，メソッドの相互の依存関係を考慮に入れたテストを行わなければならない[4]．このことから，テストに際して，テストドライバは，すべてのメソッドを任意の順序で呼び出せる必要がある．

### 2.3 テストの自動的な実行

テストは何度も反復実行される．テストの途中でバグが発見されると，バグ修正を行った後，それまでに実施したすべてのテストを反復する必要がある．また，たとえば全文テストなどで，最初から全文を網羅するようなテストケースを準備するのは容易ではない．そこで，実際に網羅率を測定し，欠けている部分について，新たなテストケースを追加することになる．バグが発見された場合にも，同じことが必要になる．バグ修正のために，すでに網羅されていた箇所に，新たな分岐が導入され，網羅率が低下することになるかもしれない．したがって，その場合にもテストケースの追加が必要になる．いずれの場合でも，テスト対象プログラムを，テストのために変更することは避けなければならないし，テストドライバやスタブも，可能な限り無修正ですませたい．

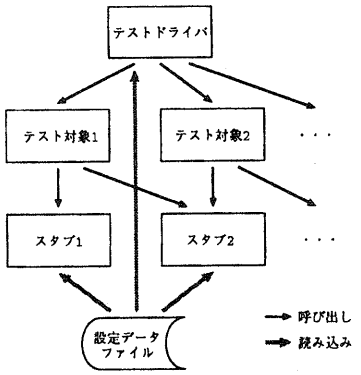


図1 テスト実行プログラムの構成

### 2.4 テスト実行プログラム

以上に述べたことから，テスト実行プログラム（ドライバ，スタブなどを含む）は以下のような方針で実現する（図1）．

(1) テスト対象コンポーネントに手を加えずにテストを実行するために，テストドライバは，テスト対象コンポーネントへの入力パラメータを自由に設定できるようにする．

(2) 同様に，スタブも，その出力パラメータ（すなわち，テスト対象コンポーネントへの入力）を自由に設定できるようにする．

(3) テストドライバは，複数のメソッド（関数）を自由な順序で何度も反復して実行できる．

(4) 以上のパラメータ設定ならびにメソッド呼び出しはすべて，設定データファイル中の記述によって設定できる．

(5) 簡単化のために，設定データファイルは一つですませたいが，誤って他のデータを読み込んで暴走することがないように配慮する．

## 3. 例　　題

IFA(Inside File Access) コンポーネントは，可変長の文字列を，インデックスを指定して，ファイルへの書き込み／読み出し／更新を行う機能を持つ．IFA は，本学で開発が進められている Inside Prolog[5] で書かれたプログラムに対して，簡易なデータベース機能を提供することを意図しているそのため，Inside Prolog 処理系

表1 IFA プログラムサイズ

| ヘッダ部 | |
|---|---|
| array.h | 31 |
| error.h | 28 |
| ifa.h | 49 |
| ifa_file_name.h | 27 |
| schema.h | 29 |
| server.h | 23 |
| session.h | 30 |
| system_call.h | 27 |
| table.h | 39 |
| tableElem.h | 34 |
| xact.h | 31 |
| ヘッダ部合計 | 348 |
| 実現部 | |
| array.c | 228 |
| error.c | 70 |
| schema.c | 311 |
| server.c | 154 |
| session.c | 136 |
| table.c | 664 |
| tableElem.c | 65 |
| xact.c | 329 |
| 実現部合計 | 1957 |
| 合計 | 2305 |

と同じC言語で実現している．ただし，設計はオブジェクト指向に基づいており，C++のクラスと1対1に対応するようにコーディングされている．このコンポーネントの開発に際しては，2で述べた方針に従って，テストドライバ，スタブを作成し，テストを実施した．その結果，すべての関数について，全文網羅ならびに全分岐網羅を達成することができた．ただし，全文網羅はtcovコマンドによって自動的に計測したものであるが，全分岐網羅は，与えたテストデータからの推定によるものである．

上記のテストドライバならびにスタブはすべて人手で作成したが，このようなテストドライバ，スタブの自動生成を目的として，今回，人手で作成したプログラムを分析，整理した．

テスト対象となったIFAコンポーネント自体のプログラムサイズを表1に，テスト用プログラムのサイズを表2に示す．表2の`array.c`と`error.c`は，他のモジュールをテストするために用いるもので，単なるスタブではなく，スタブと本来のモジュールとを合体したものになっている（5参照）．ファイル`*_drv.c`には，各モジュールのテストプログラムの`main( )`関数がある．ファイル`*_stb.c`には，いわゆるスタブの他に，パラメータ設定のためにドライバ（`main( )`）から呼び出される関数が含まれている（6参照）．

## 4. パラメータの分析

テスト対象コンポーネントに対して，ドライバやスタブから渡すパラメータの設定を変更することによって，コンポーネント内部の実行パスや機能をテストすることができる．原則として，パラメータの設定次第で，すべての実行パスや機能をテストできるはずである☆1．さもなければ，コンポーネント中にデッドコード（決して実行されることのないコード）があったり，決して発現しない機能を持っていることになる．

表2 IFA 用テストプログラムサイズ

| ヘッダ部 | |
|---|---|
| array_stb.h | 29 |
| schema_stb.h | 36 |
| server_stb.h | 33 |
| session_stb.h | 37 |
| tableElem_stb.h | 31 |
| table_stb.h | 54 |
| xact_stb.h | 43 |
| ヘッダ部合計 | 263 |
| 実現部 | |
| array.c | 277 |
| array_drv.c | 52 |
| array_stb.c | 396 |
| error.c | 75 |
| schema_drv.c | 78 |
| schema_stb.c | 496 |
| server_drv.c | 45 |
| server_stb.c | 420 |
| session_drv.c | 56 |
| session_stb.c | 506 |
| system_call.c | 166 |
| tableElem_drv.c | 50 |
| tableElem_stb.c | 257 |
| table_drv.c | 118 |
| table_stb.c | 713 |
| xact_drv.c | 90 |
| xact_stb.c | 673 |
| 実現部合計 | 4468 |
| 合計 | 4731 |

パラメータとしては，記述言語が許すあらゆる型のデータを渡すことができるが，現実のプログラムにおいてはそれほど多様性があるとは考えられない．今回例題として用いたIFAの8個のファイル☆2中の33個の関数☆3のパラメータ（引数）の型を調べたところ☆4，表3に示すように整理することができた．

以下，それぞれの型について検討する．

### 4.1 int 型

任意のint型の値を渡すパラメータである．IFAには登場しなかったが，char型，float型などでも同様である．テストデータとして，具体的にどのような値を与

☆1 並行処理，例外処理，割り込みなどは対象外としている．
☆2 C++のクラスに相当．
☆3 C++のメソッド（メンバ関数）に相当．
☆4 Prologプログラムとのインタフェース部は別プログラムになっており，今回の例題には含まれていない．したがって，インタフェースはC言語における通常のやり方に従っている．

表3 IFAにおける関数のパラメータの型

| 分類 | 型 | 備考 |
|---|---|---|
| 基本型 | int | enumを含む |
| 派生型 | (なし) | ポインタ以外 |
| ポインタ型 | void * | |
| | int * | |
| | char * | |
| | char ** | |
| | struct 構造体名 * | |
| | struct 構造体名 ** | |

えるべきかは，コンポーネントの仕様や，ソースコードの構造に依存する．機能仕様やソースコードから自動的にテストデータを生成することは容易ではなく，一般にはテストのための仕様を与える必要がある．本研究では，テスト担当者が人手で判断し，値を設定するものとする．もちろん，設定はファイル中で行うので，1度設定すれば，何度でも再利用できる．設定した値が適切であったかどうかは，どのようなテストを行っているのかに依存するが，全文テスト，全分岐テスト，データフローテストに関しては，ツールで自動的に結果を測定できるので，意図した通りのテストが実行されたかどうかを確認することができる．十分な網羅が達成されていない場合には，データの変更や追加を行うことになる．

### 4.2 ポインタ以外の派生型

ポインタ以外の派生型を引数として受けとる必要はほとんどない．別の関数が返す派生型の一時オブジェクトをそのまま引数として用いる場合，一時オブジェクトのアドレスを受けとれないので，派生型へのポインタではなく，派生型を引数として受けとる必要があるが，きわめて例外的な状況である．これに関しても，C++ではconstを指定することによって，参照引数とすることができる．

### 4.3 void * 型

任意の型のポインタを受けとるために用いられる．しかし，このポインタを用いてその先のオブジェクトへアクセスすることはできない．ただ単にそれを受けとって，どこかへコピーしたり，保存したりするだけである．したがって，テストのためには，ポインタと同じ大きさの整数値を渡すだけでよい．

### 4.4 int * 型

通常は，整数値を呼び出し側へ返すために用いる．したがって，ドライバは，整数型の変数（領域）を準備し，そのアドレスを渡せばよい．ただし，すでにそこに値が入っていて，それを更新する場合もある．そこで，いくつかの同じ，あるいは異なる関数の呼び出しにまたがって，その整数型変数の値を保持する機構が必要である．もっともよく起こり得るエラーは，ポインタとしてNULLが渡される場合である．ドライバ側で，NULLの設定もできなければならない．ポインタがNULLではないが，無効な値となることは，コンパイラによる型チェックが守られている限り起こり得ないので，そのような状況をテストする必然性は小さい．

### 4.5 char * 型

文字列を渡す，領域へのポインタを渡して文字列を返す，渡された文字列を更新する，などがありうる．今回調査した例題では，すべて文字列を渡す場合であった．領域へのポインタを渡す場合や，文字列を更新する場合，領域の大きさが問題となり，エラーが発生しやすいので，そのような場合についてもテストしなければならない．ただし，今回は考慮していない．

### 4.6 char ** 型

新たに割り付けた文字列（あるいは領域）を返す，静的な文字列へのポインタを返す，などがありうる．前者の場合には，呼び出し側でその文字列を解放しなければならない．一方後者は，逆に，解放してはならないし，原則として更新もあってはいけない．そのような観点からは，この型のパラメータを用いるべきではないが，IFAでは，2箇所に出現していた．この問題は，呼び出し側に要求されることであって，テスト対象コンポーネントに要求されていることではなく，どのようなテストを行うかには，直接は関係しない．テストのためには，char *型の領域を準備して，そのアドレスを渡すか，あるいはNULLを渡せばよい．

### 4.7 struct 構造体名 * 型

一般に，構造体をパラメータとして渡す場合の型である．構造体は，単に領域が確保されていればよい場合と，適切な値が設定されていなければならない場合とがある．ドライバやスタブで構造体に適切な値を設定するのは容易ではない．テスト対象コンポーネント中に，構造体の初期設定を行う関数があれば，それを使えばよい．そうでない場合には，別のコンポーネント中に，構造体の初期設定を行う関数があるはずである．そこでそのコンポーネントのテストを先にすませ，それをテストに用いればよい．ただし，5において述べるシステムコールやライブラリの場合と同様に，テスト対象コンポーネントとの間にインタフェーススタブを用意するか，または，そのテストの済んだコンポーネントを改造してスタブを作成する必要がある☆．IFAでは，後者の書換えの手法を採用していたが，スタブ作成時にオリジナルの部分にバグを埋め込んでしまわないように十分注意を払う必要があり，むしろ，システムコールやライブラリと同じ手法をとる方が望ましい．

### 4.8 struct 構造体名 ** 型

新たに割り付けた構造体（あるいは領域）を返す．char **型の場合と同様，望ましくないパラメータ型ではあるが，IFAでは，6箇所に出現していた．ただし，いずれの場合でも，同一コンポーネントの別の関数で，割り付けられた構造体を解放することになっており，char **型ほどの混乱はない．テストは，char **型と

☆ このコンポーネントはテスト対象コンポーネントではないので，書き換えてもよい．

表4　文字列パラメータの設定例

| データ | | 設定値 |
|---|---|---|
| 0 | ? | NULL ポインタ |
| 1 | ? | 長さ0の文字列 |
| 1 | name | 文字列 "name" |

同様に，struct 構造体名 * 型の領域を準備して，そのアドレスを渡すか，あるいは NULL を渡せばよい．この場合には，構造体の初期化について心配する必要はない．

## 5. 設定データファイルの形式

設定データファイルは，ドライバやすべてのスタブで，同一ファイルを順に読み出すものとする．これによって，関数間の制御の流れがテストの意図通りであることを確認することになる．一方で，ファイル中に必要な数値や文字列だけを記載するという方法では，誤ったデータの読出しや，偶然の一致から別の変数への入力データと解釈される可能性がある．そこで，設定データファイルの形式を図2のように定めている．

(1) **テストケース番号**：テストケースの番号を記す．番号は任意で，異なるテストケースが同一の番号を持ってもよい（チェックしていない）が，同一のテストケースは同じ番号でなければならない．どれだけのものを一つのテストケースにするかは，テスト担当者にまかされており，自由に設定できる．後述の **set_test_case( )** によって，新しいテストケースを設定する．例えば，一つのクラスオブジェクトの生成から破棄までを一つのテストケースとする．

(2) **関数名**：ドライバやスタブ中のすべての関数は，必ずこのデータファイルを読み込む．このときに読み込んだ関数名が自身の関数名であることを確認する．システムコールやライブラリに関しても，一旦スタブを経由することによって，データファイルを読み込み，実行経路を確認する．

(3) **パラメータ** ドライバがテスト対象コンポーネント中の関数を呼び出すときのパラメータや，スタブが呼出側へ返すパラメータなどを設定するために，数値や文字列を読み込む．パラメータの型や数は関数に依存する．文字列は，具体的な文字列を設定する場合の他に，NULL ポインタを設定する場合や，長さ0の文字列を設定する場合がある．これを一つの文字列で表現することはできないので，整数値と文字列の二つのデータをファイルから読み込んで一つの文字列パラメータを設定する．設定の例を表4を示す．ただし，この設定は一例であり，一般化にはさらに検討が必要である．

設定データファイルの一部を図3に示す．

```
0 set_test_case 1
1 next_call 1
1 set_parm_new 0 0
1 error_error
1 eval_result_new
1 next_call 0
1 set_test_case 2
2 next_call 1
2 set_parm_new 0 1
2 malloc -1
2 error_error
2 eval_result_new
...
```

図3　設定データファイルの例

## 6. テスト実行プログラムの実現

テストドライバの実行部の例を図4に示す．**test_start( )** は，設定データファイルのオープンや，変数の初期化を行い，**test_end( )** は終了処理を行う．**set_test_case( )** は，テストケース番号を更新する．他の関数は，設定データファイルから読み込んだテストケース番号が，現在の番号と一致することを確認する．

基本的な制御は，関数 **next_call( )** によって行われる．**next_call( )** は，あるテスト対象関数の呼び出しの系列，すなわち，パラメータ設定（**set_parm_Server_login( )** など），関数呼び出し（**ifa_login( )** など），結果の評価（**eval_result_Server_login( )** など）の系列が終了する度に呼び出され，次に呼び出すべきテスト対象関数を指定する．**next_call( )** のコードの例を図5に示す．**set_test_case( )** も，2回目以降は，**next_call( )** の指定によって呼び出される．

パラメータ設定や結果の評価，ならびにスタブも，**next_call( )** と同様に設定データファイルを読み込み，パラメータの設定を行う．結果の評価ではパラメータ設定の必要はないが，制御フローの確認のために設定データファイルを読み込む．

テスト対象関数から呼び出される他のモジュールの関数の一部や，システムコールは，テストケースによって，ダミー処理がふさわしい場合と，本来の処理が必要な場合とがある．たとえば，記憶域を割り付ける関数 **malloc( )** を呼び出したとき，十分な記憶域が確保できずに，NULL が返される場合についての対処がきちんと行われているかどうかをテストするには，実際に **malloc( )** を呼び出すよりも，スタブで無条件に **NULL** を返す方が簡単である．一方，記憶域を確保できた場合のテストも必要であり，スタブだけで対応することはできない．そこでこの場合も，設定データファイルを読み込み，ファイルから読み込んだパラメータに従って両者を切り替える．

| 番号 | 関数名 | パラメータ1 | パラメータ2 | ・・・ |
|---|---|---|---|---|

番号： テストケース番号
関数名： この行を読み出す関数
パラメータ： 設定すべきパラメータ．
関数によって数が異なる．

図2 設定データファイルの形式

```
test_start( );
set_test_case( );
while( (t_nextCall = next_call( )) >= 0 ) {
  switch(t_nextCall) {
  case RESTART:
    set_test_case( );
    break;
  case LOGIN:
    set_parm_Server_login(&t_DBNameP, &t_userIDP, &t_passwdP, &t_sessionPP,
                          &t_errorCodeP);
    t_retCode = ifa_login(t_DBNameP, t_userIDP, t_passwdP, t_sessionPP,
                          t_errorCodeP);
    eval_result_Server_login(t_retCode, t_sessionPP, t_errorCodeP);
    break;
  case LOGOUT:
    set_parm_Server_logout(t_sessionPP, &t_errorCodeP);
    t_retCode = ifa_logout(*t_sessionPP, t_errorCodeP);
    eval_result_Server_logout(t_retCode, *t_sessionPP, t_errorCodeP);
    break;
  }
}
test_end( );
```

図4 テストドライバの実行部の例

## 7. 自動生成

テストドライバやスタブのプログラムサイズは決して小さくない．IFAの場合，表1，表2からもわかるように，テスト用プログラムの行数の合計は，テスト対象プログラムの行数の2倍を越えている．テストドライバの一部である`next_call( )`のコードを図5に示しているが，次にテストすべき関数を指示するための整数値を一つ読み込むだけで，これだけの処理が必要となる．この図に示したコードは，テストドライバやスタブ中のすべての関数にほぼ共通のコードである．

それぞれのテスト用の関数が異なるのは，おもにパラメータの取り扱いである．4及び5で述べたように，IFAに関しては，パラメータの型によって，一意に処理方法を決定することができる．したがって，テスト対象関数ならびに，それが呼び出す関数のインタフェース仕様を与えることによって，テストドライバやスタブを自動生成することができる．

より一般的なプログラムを対象とする場合，処理方法が一意に決まらない可能性があるが，それでもいくつかのパターンに分類できると予想している．そこで，テスト担当者に，どのパターンに該当するかを指定してもらうことによって，自動生成が可能になる．その場合でも，一般的なテスト仕様を記述するよりははるかに少ない記述量ですむものと思われる．

これまでに，いくつかのスタブ関数について，Tcl/Tkを利用して，インタフェース仕様から必要なスタブ関数を自動生成できることを確認している．今回の分析結果に基づいて，今後必要なすべての関数を自動生成するシステムを試作する予定である．

## 8. おわりに

著者らはデータフローテスト支援ツール[6]の開発も進めている．今回調査したIFAの一部である`array.c`に関して，データフローテスト基準を調べたところ，いくつかの関数では100%網羅に達していないことが判明した．これについて，単にテストケースを追加するだけで100%網羅が達成できるかどうかは，今後検討する必要がある．また，オブジェクト指向への対応として，一歩進んだデータフローテスト基準[4]を提案しているが，まだツールとして実現されておらず，今後の課題である．

## 参考文献


1) Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.
2) Sun Microsystems. *SunOS Reference Manual,* 1988.

```
int
next_call( )
{
  int   t_caseNo;
  char  t_funcName[BUFSIZ];
  int   t_retVal;

  if( fgets(ifa__lineBuf, BUFSIZ, ifa__inputFile) == NULL ) {
    fprintf(stderr, "next_call: cannot read file: %s\n", ifa__inputFileName);
    exit(-1);
  }
  ++ifa__lineNo;
  if( sscanf(ifa__lineBuf, "%d%s%d", &t_caseNo, t_funcName, &t_retVal)
     != 3 ) {
    fprintf(stderr, "next_call: illegal file: %s\n", ifa__inputFileName);
    fprintf(stderr, "%d: %s", ifa__lineNo, ifa__lineBuf);
    exit(-1);
  }
  if( t_caseNo != ifa__caseNo ) {
    fprintf(stderr, "next_call: illegal test case number: %s\n", ifa__inputFileName);
    fprintf(stderr, "%d: %s", ifa__lineNo, ifa__lineBuf);
    exit(-1);
  }
  if( strcmp(t_funcName, "next_call") != 0 ) {
    fprintf(stderr, "next_call: illegal stub function name: %s\n", ifa__inputFileName);
    fprintf(stderr, "%d: %s", ifa__lineNo, ifa__lineBuf);
    exit(-1);
  }
  if( t_retVal > MAXCOUNT ) {
    fprintf(stderr, "next_call: illegal return value: %s\n", ifa__inputFileName);
    fprintf(stderr, "%d: %s", ifa__lineNo, ifa__lineBuf);
    exit(-1);
  }
  return t_retVal;
}
```

図5 next_call のコード例


3) Shel Siegel. *Object-Oriented Software Testing: A Hierarchical Approach.* John Wiley & Sons, 1996.

4) 廣田 豊彦, 橋本 正明. オブジェクト指向プログラムのためのデータフローテスト. 電子情報通信学会論文誌 *D-I*, J97-D-I(10):707–718, 1996 年 10 月.

5) Keiichi Katamine, Masanobu Umeda, Neng-Fa Zhou, and Isao Nagasawa. Design and implementation of integrated development environment for knowledge-based system. In *INAP '97: The 10th Symposium on Industrial Application of PROLOG, 1997*, pages 28–35, October 1997. (In Japanese).

6) T. Hirota and M. Hashimoto. Integrated testing environment for C++. In *Proceedings of Japan-CIS Symposium on Knowledge Based Software Engineering '94*, pages 120–125, May 1994.