

ソフトウェア工学: テスト技法

(Software Testing)

Behrouz H. Far

埼玉大学情報システム工学科

Department of Information and Computer Sciences,

Saitama University

目次

1. 定義

- ソフトウェア有効性 (Validation)
- ソフトウェア検証 (Verification)
- ソフトウェアテスト (Test)

2. テスト概念

3. テスト方法

- 机上テスト (Theoretical test)
- マシンテスト (Machine test)

4. テストケース設計技法

- ブラックボックステスト
- ホワイトボックステスト

5. テスト手法：その他

6. 結び

定義

- ソフトウェア有効性 (Validation):

ソフトウェアシステムはスペックと一致しているかどうかを確認する。

- 例 :

「Requirement Review」

定義

- ソフトウェア検証 (Verification):

各プロジェクトフェーズの出力は入力によって正しくとりだされてるかどうか、及び、正しくつなぎ合わせてるかどうかを確認する。

- 例 :

「Unit Testing」

定義

- ソフトウェアテスト:

ソフトウェアテストとは、プログラムの正しさを確認し、エラーがないことを示すこと。具体的に:

- ソフトウェアテストは「プログラムコード」を対象にするソフトウェア有効性 (Validation) とソフトウェア検証 (Verification) の一部である。

- プログラム設計フェーズテスト
- オペレーションフェーズテスト
- メンテナンスフェーズテスト

テスト概念

P: プログラム

F: スペック

D: テストデータ

x: プログラム入力

$$\forall x \in D \mid P(x) = F(x) \implies \forall x \mid P(x) = F(x) \quad (1)$$

逆に,

$$\exists x \mid P(x) \neq F(x) \implies x \in D \quad (2)$$

これを確実に確かめることができない.

テスト方法: 机上テスト

机上テスト = 机上デバッグ = 静的テスト

プログラムをマシンで実行する前に検査する.

- ウォークスルー (walk through)
- インスペクション (inspection)

Specifi-
cation



Source
Program

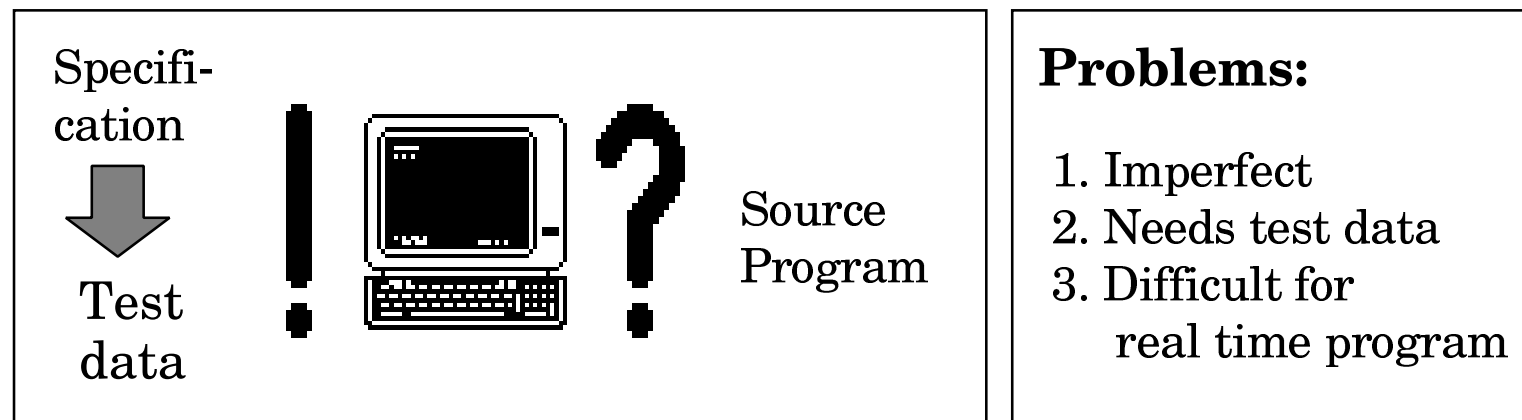
Problems:

1. Expensive
2. time consuming
3. human errors

テスト方法: マシンテスト

マシンテスト = マシンデバッグ = 動的テスト

プログラムをマシンで実行しながら動作状態を調べる。



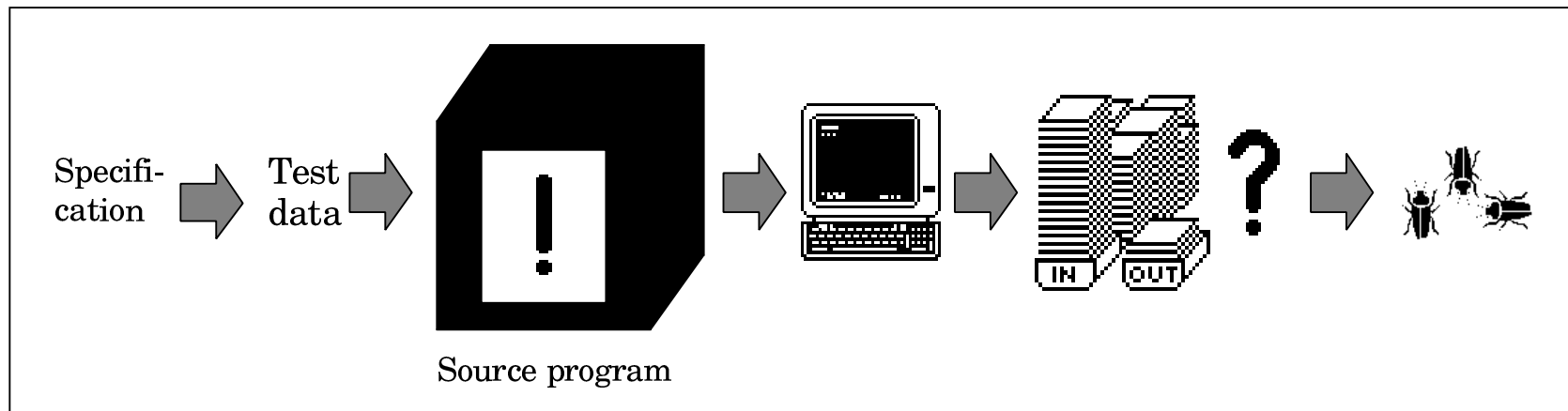
マシンテストの手順

1. 机上テストよりテストケースを設計する.
2. テストケースには, 期待される出力結果も含めて考慮する.
3. ソースプログラムをコンパイルし, 実行形式のロードモジュールを作成する.
4. テストケースには, ファイル入出力を取り扱う場合, ディスク上にファイルの領域確保をする.
5. ディスク上に, テストケースに対応するデータを格納する.
6. ロードモジュールの実行環境を整える.
7. ロードモジュールを実行させ, 出力結果を得る.
8. 出力結果を確かめ, 期待された出力通りかどうかチェックする.
9. 5 - 8 の作業を繰り返す.

ブラックボックステスト

プログラムの外部仕様にもとづいてテストケース設計 (WHAT ?)

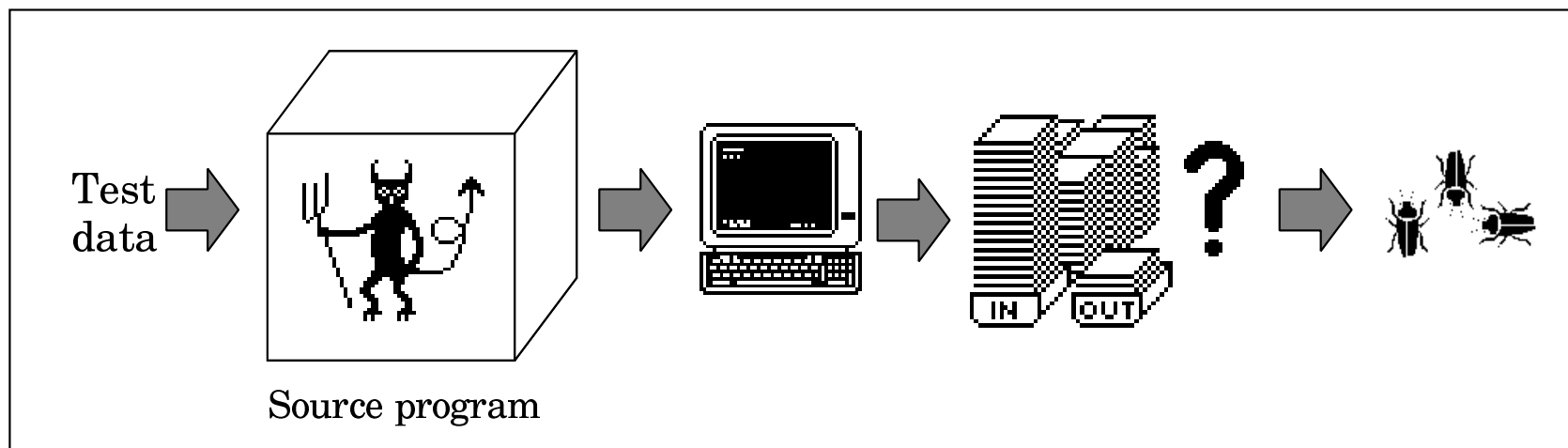
- プログラムのロジック経路はまったく考慮しない.
- 仕様の確認をとるための基準が不明確となりやすい.
- コーディングミスが発見しにくい.



ホワイトボックステスト

プログラムの内部仕様にもとづいてテストケース設計 (HOW ?)

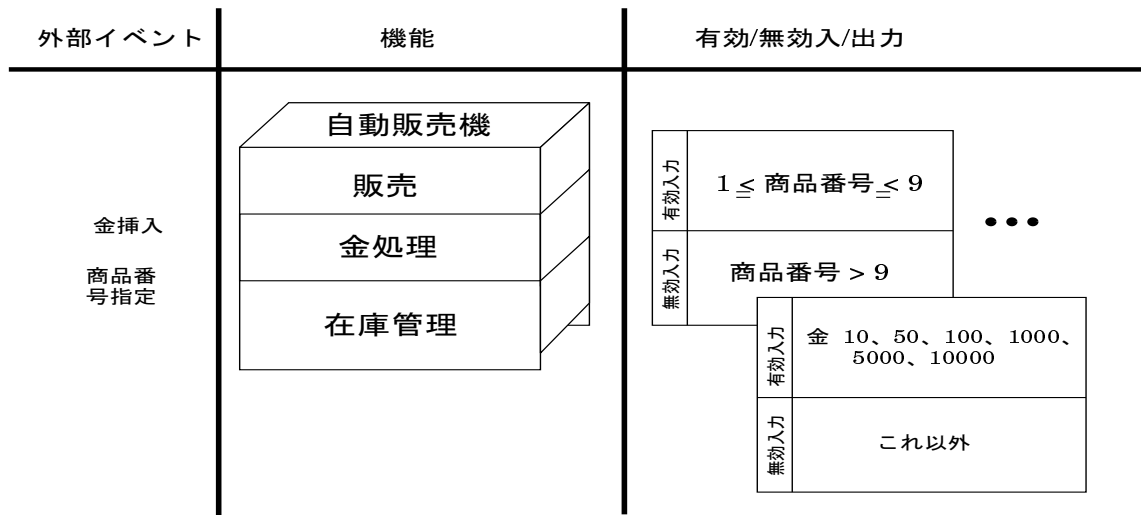
- プログラムのロジックにもとづくテストケースを設計
- プログラムの全ステートメントを通るようにテストスイットを設計
- プログラムのすべての分岐 (枝) をテストすることが不可能約 %85 分岐 (枝) を通るようにはテストスイットを設計



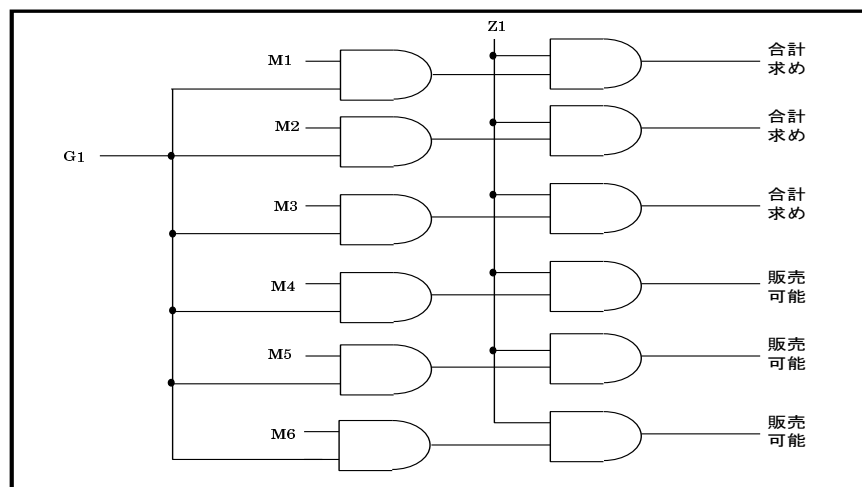
ブラックボックステスト

1. 機能テスト (Functional testing)
2. ランダムテスト (Random testing)
 - システムスペックにもとづくテスト:
システムスペックのもとにテストを行なう。IPO 図及び PAD を使う。
 - システム設計にもとづくテスト:
システムのプログラムモジュールをもとにしてテストを行なう。

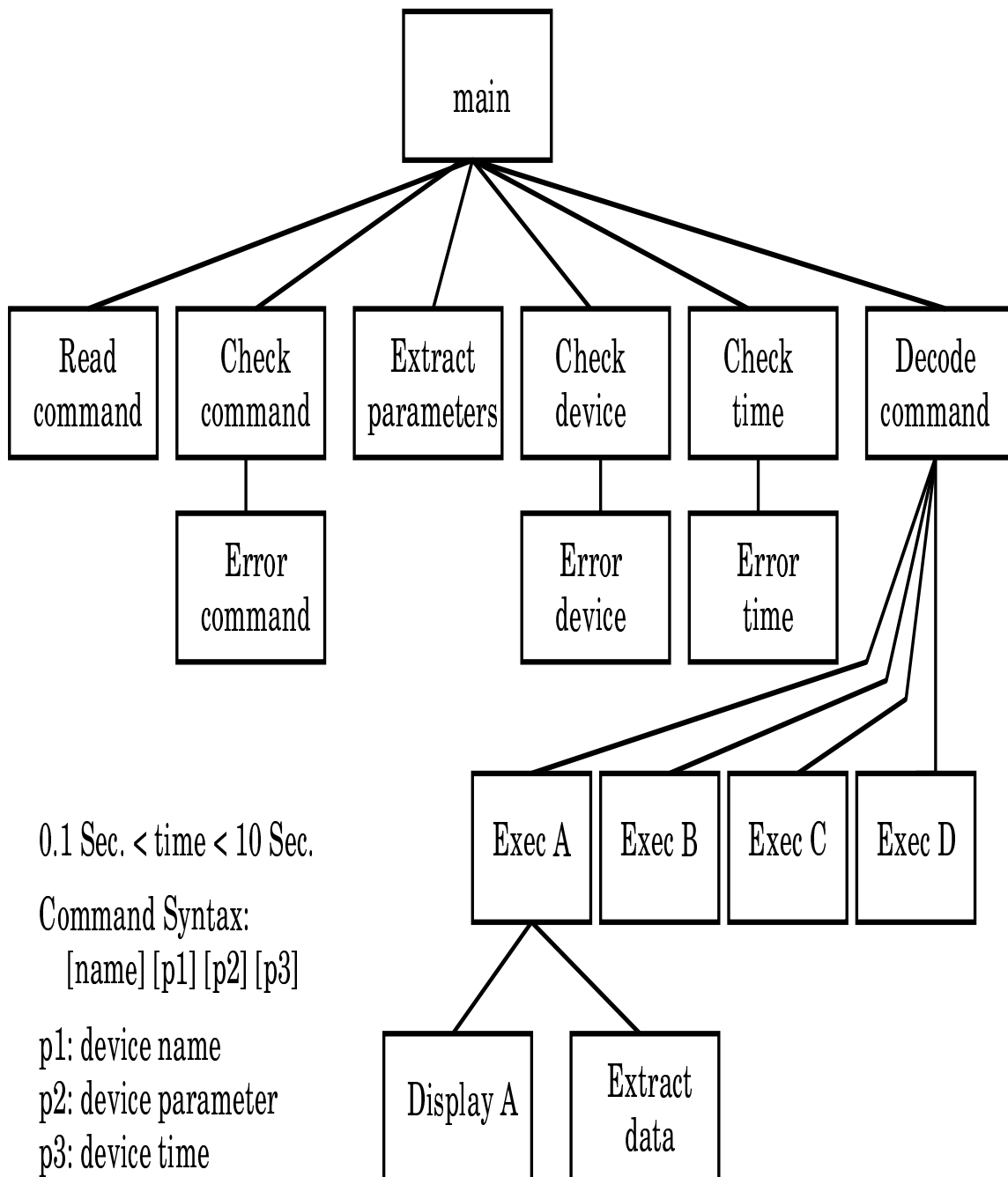
ブラックボックステスト： システムスペックにもとづくテスト



		M1	M2	M3	M4	M5	M6							
	金挿入	10	50	100	1000	5000	10000							
G1	商品番号 1 - 9	合計 求め	合計 求め	合計 求め	在庫 求め	在庫 求め	在庫 求め	<table border="1"> <tr> <td>Z1</td> <td>Z2</td> </tr> <tr> <td>在庫 十分</td> <td>在庫 不十分</td> </tr> <tr> <td>販売 可能</td> <td>販売 不可能</td> </tr> </table>	Z1	Z2	在庫 十分	在庫 不十分	販売 可能	販売 不可能
Z1	Z2													
在庫 十分	在庫 不十分													
販売 可能	販売 不可能													
G2	商品番号 > 9	—	—	—	—	—	—							



ブラックボックステスト： システム設計にもとづくテスト



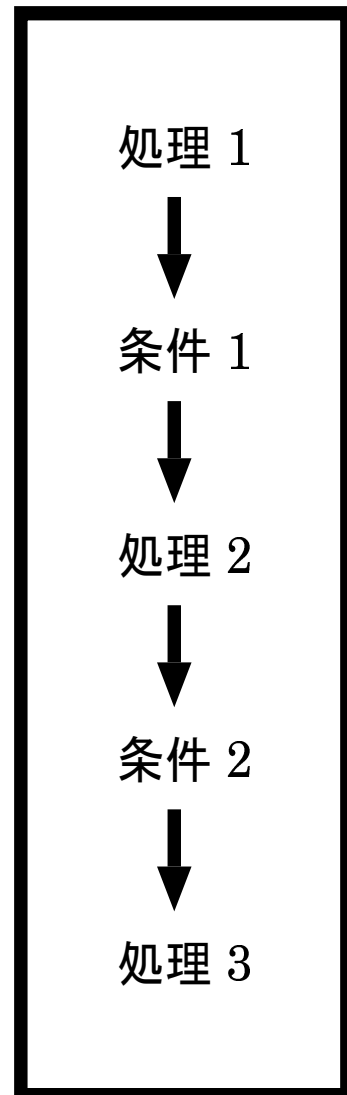
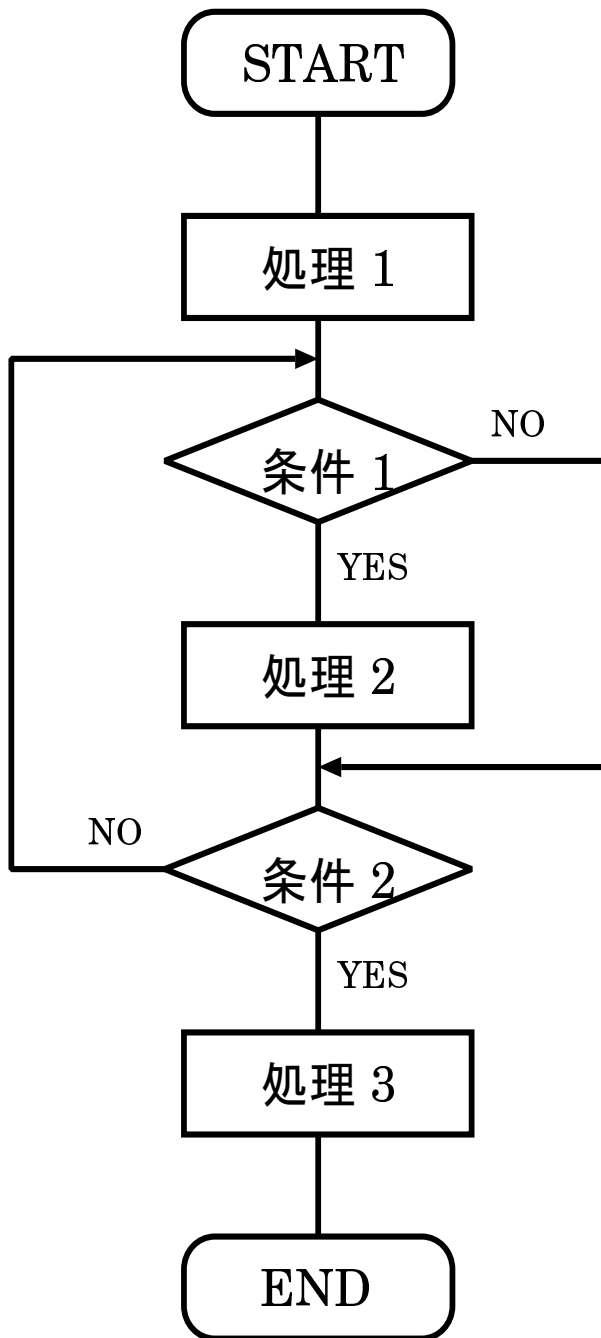
ホワイトボックステスト

ホワイトボックステスト = 構造テスト

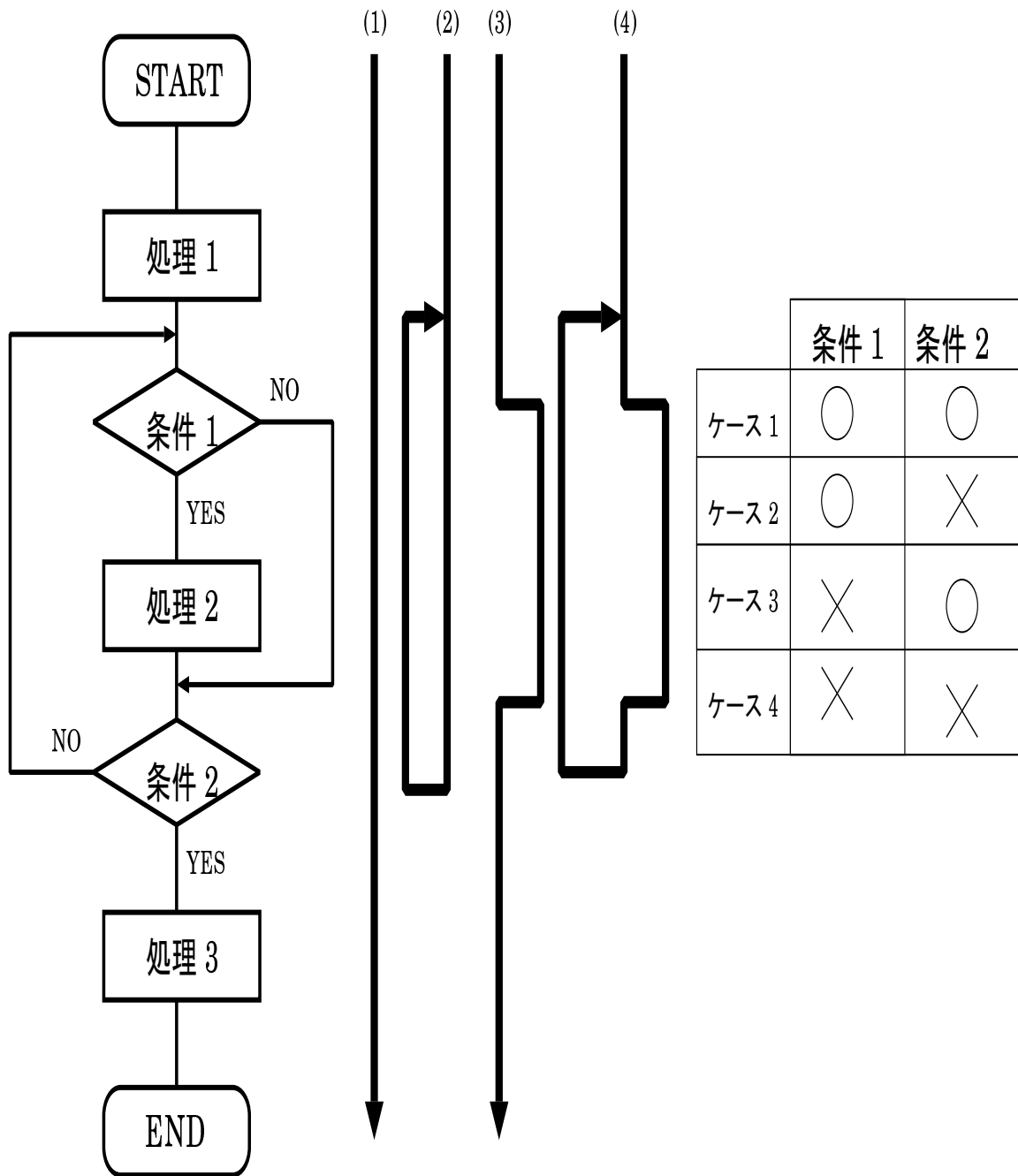
プログラムのアルゴリズムである内部構造を見ながら、ロジック仕様書やプログラムリストもとにしてテストスイートを設計し、テストを実行する。

1. 命令網羅
2. 分岐網羅
3. 条件網羅
4. 条件-分岐網羅

ホワイトボックステスト：命令網羅



ホワイトボックステスト：分岐網羅



テスト手法：その他

- Mutation Testing
- Random Testing
- Structural & Data flow-oriented Testing
- Assertion & Adaptive Testing
- Grammar-based Testing
- Symbolic Testing
- Domain Testing and input space partitioning
- Functional Testing
- Cause-effect Testing

Mutation Testing

1. 元のプログラムの一部を変化する。変化された部分を「ミュータント (mutant)」と名付ける。
 2. ミュータント集合を用意する。
 3. テストを行って、元のプログラムとミュータントの出力を比較する。
 4. 元のプログラムとミュータントの出力は一致しなければそのミュータントを捨てる。
 5. 全てのミュータントは消えるまでに 3～4 を繰り返す。
- **長所**： テストデータが十分かどうかを確認できる。
 - **短所**： ミュータント数が多い。

Random Testing

1. プログラムの各入力の値の範囲を明確にする。
 2. その範囲内のテストデータを生成する。
 3. テストデータによる出力を生成する。
 4. 出力データは正しいかどうかを確認する。
- **長所**： テストデータ生成しやすい。
 - **短所**： プログラム構造によるのバグを発見できない。

Assertion Testing

- 変数値の間にチェックコード又は判別式コードを挿入する。
- 例えば、プログラム内でいつも $a + b \leq c$ を確かめるためには、次の文をプログラムに挿入する：

```
...  
...  
if (a + b > c) {  
    print ‘‘ERROR: violation of condition 124\n’’;  
}  
...  
...
```

- 長所： エラー発生しやすいプログラム部分を動的にチェックできる。
- 短所： ソフトウェアテストツールを使う。

Symbolic Testing

- 普段は、テストデータによって、プログラム出力を生成するが、シンボリックテストの場合は、テストデータによるプログラムのシンボリック出力を生成する。
- 例えば、次の pascal コード

```
...  
i:=1;s:=0;  
while i < n do  
  begin  
    s:=s+a[i];  
    i:=i+1  
  end;  
print(s)  
...
```

のシンボリック出力 (n=4) の場合は、
 $a[1] + a[2] + a[3] + a[4]$
になる。

- **長所**： テストケースは自動的に生成される。

Domain Testing

- 入力集合をいくつかのサブセットに分けて各サブセットによってプログラムを部分的にテストする。
- あるサブセットによるプログラムをテストする時、他のサブセットの変数が変わらぬと考える。

- **長所**： テストリソースが少ない。
- **短所**： オーバーラップ部分のバグは発見しにくい。

Cause-Effect Testing

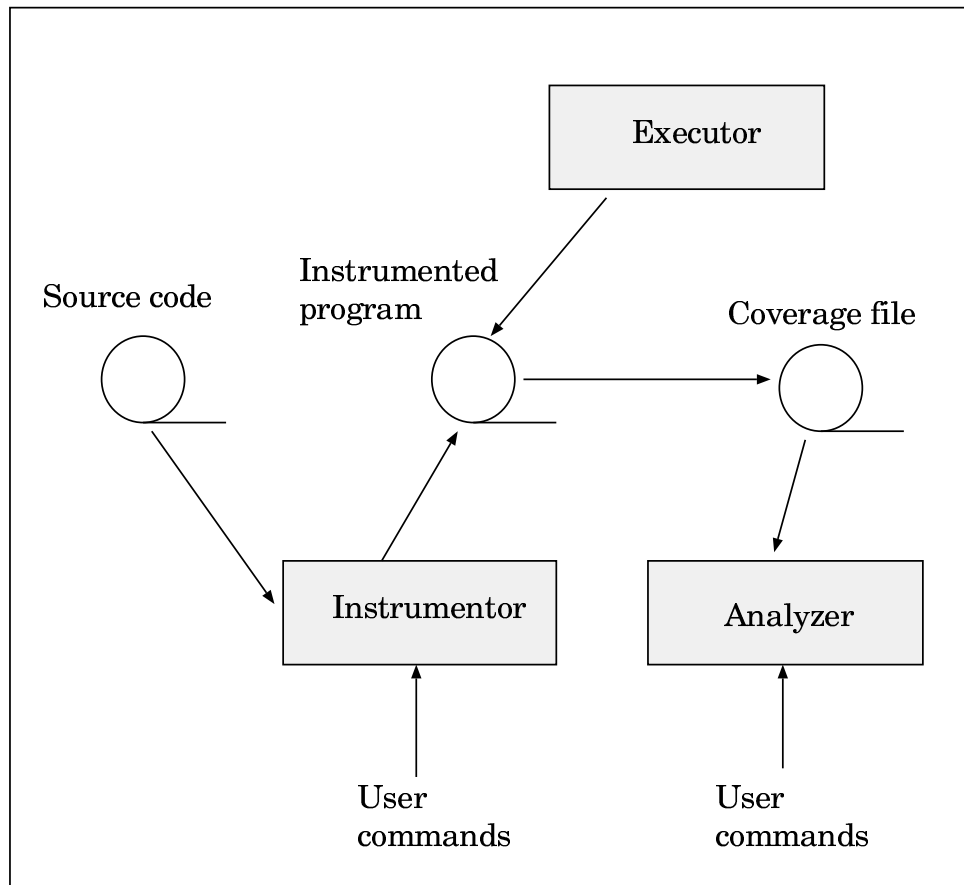
- 入力変数間の因果関係を明確にする
- 「Causal Ordering」 手法によって入力変数のサブセットを生成する。
- 下位の階層の変数に起きた摂動が上位の階層にどのように影響していくかを解析する。

- **長所**： 複数の原因によるバグが発見しやすい。
- **短所**： SE 以外は AI 技術が必要である。

ソフトウェアテストツール

- テストスイート設計：仕様記述言語 (Specification and Description Language: SDL)
- テスト実行・シミュレーション：Instrumentor
- GNU テストツール：Dejagnu

Instrumentation



"Instrumentor" inserts program code into a unit under test.

Original program :

```
procedure check_temp (temp:integer; Var flag:boolean)
begin
if temp > 100 then flag := true
else flag := false
end;
```

Instrumented program :

```
procedure check_temp (temp:integer; Var flag:boolean)
begin
writeprocname('check_temp')
if temp > 100 then
begin
writebranch('check_temp', 1); flag := true
end
else
begin
writebranch('check_temp', 2); flag := false
end
end;
```

SDT Case Tool

