

Debug Engineering

JaSST' 13TOKAI

論理の設計とテストを考える ＜CFD++、その目指すところ＞

2013年11月

(有)デバッグ工学研究所 堀田

R-13.9

■ 目次

はじめに

1. CFDとは

～原因流れ図(CFD)の狙いと方法～

2. CFD++

～CFDの進化の方向～

2.1 設計段階での制約

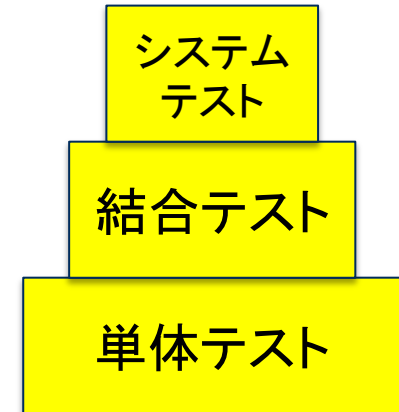
2.2 Concollic Testingとの融合

■ テストは依然、大きな問題

- 開発工数の50%以上を占めることも

■ テストは積み上げ

- 単体→結合→システム (など)
- 適切なテスト区分が大切



■ テストへの要求

- 漏れなく
- 効率的に

→ 実現するためにはテスト技法

■ テスト技法使用上の問題

- 技術不足、応用できない・・・学習・訓練しかない
- 仕様書から抽出すると条件(因子、水準)が多くなりすぎる
同値分割、デシジョンテーブル、All-Pair
⇒ この問題を考える

■ 原因は？

- テスト対象範囲区分がうまくない: 単体テスト範囲、結合テスト範囲、・・・
- 仕様書の情報が曖昧

■ 背景は？

- プログラムの分割 = 構造設計が適切ではない
- テストに必要な設計情報が漏れる

プログラム＝テスト対象の構造を考える

■ プログラムは何を実現しているのか

- ビジネスルール、装置制御ルール
 - － 主に外部仕様 ①
- 実装上必要な処理: コンパイラ、ミドルウェア、OSなどの制約適合、他
 - － 開始手続き(初期設定、ログイン、..) ②
 - － 便宜的処理(中間結果出力、I/O、ループ、..) ③
 - － 実行対応(排他制御、同期、..) など ④
- その他 ⑤

■ テストの区分

- ①、②と③の一部は論理のテスト
- 他は様々 (振る舞い、官能評価、..)

■ 論理の内容

- 動作、条件と判定
- 仕様: $○○$ (条件の要素) が $△△$ の時 (条件の範囲) に $□□$ する (動作)
- プログラムでは判定が必要 $<○○$ が $△△$ の範囲にあるか $>$

■ 論理の設計とテスト

- 設計では条件、判定、動作を詳細化
- テストは設計された論理 (条件、判定、動作) が仕様通りかを確認

■ 論理テストを容易にするには

- 条件、判定、動作とその関係がわかり易いこと
(数の制限、関係の明示、..)
- 論理構造分析と適切な分割
- そのためには構造を表現すること

- 論理構造を明らかにして、決定表に展開するための技法
＝効率的に論理をテストするための方法論
- 技術者がCEG(原因結果グラフ)を容易に理解できないのを見て考えだされた。
- NEC(当時)松尾谷 徹(デバッグ工学研究所)
- 本日はその内容と発展形(今後の展開)の説明
 - CFDの考え方
 - CFDの現在と次の展開
(堀田の解釈)

1. *CFD*とは

～原因流れ図(*CFD*)の狙いと方法～

■ 原因結果グラフが普及しない原因分析から始まった

- 原因: 原因結果グラフの難しさ
 - 電子回路設計の概念を用いており、SW技術者が理解できない
 - SWの仕様は、2値論理で記述されていないので、取り扱いが難しい

■ SW技術者が用いる設計概念を応用して開発

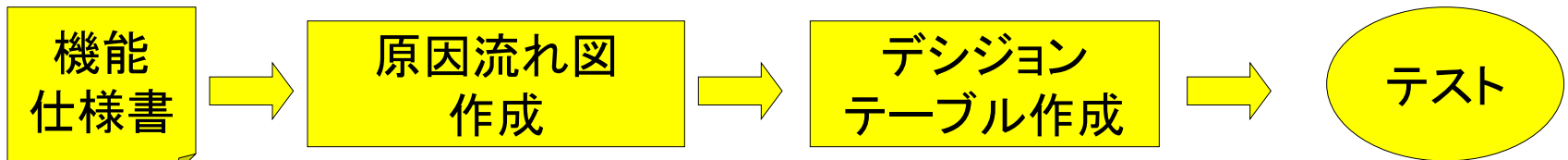
- CFDは1980年中頃、OS開発の現場で生まれた 処理の流れから、結果に関連する条件を特定する
- さらに、仕様では必ずしも定義されない、無効系の流れを追加

手順概要

■ 処理の流れ、データの流れたに沿った条件に限定表現

- 原因流れ図を書く
- テスト用のデシジョンテーブルにまとめる
- 主に結合テストの合理化

考案者 松尾谷 徹氏 (現デバッグ工学研究所代表)

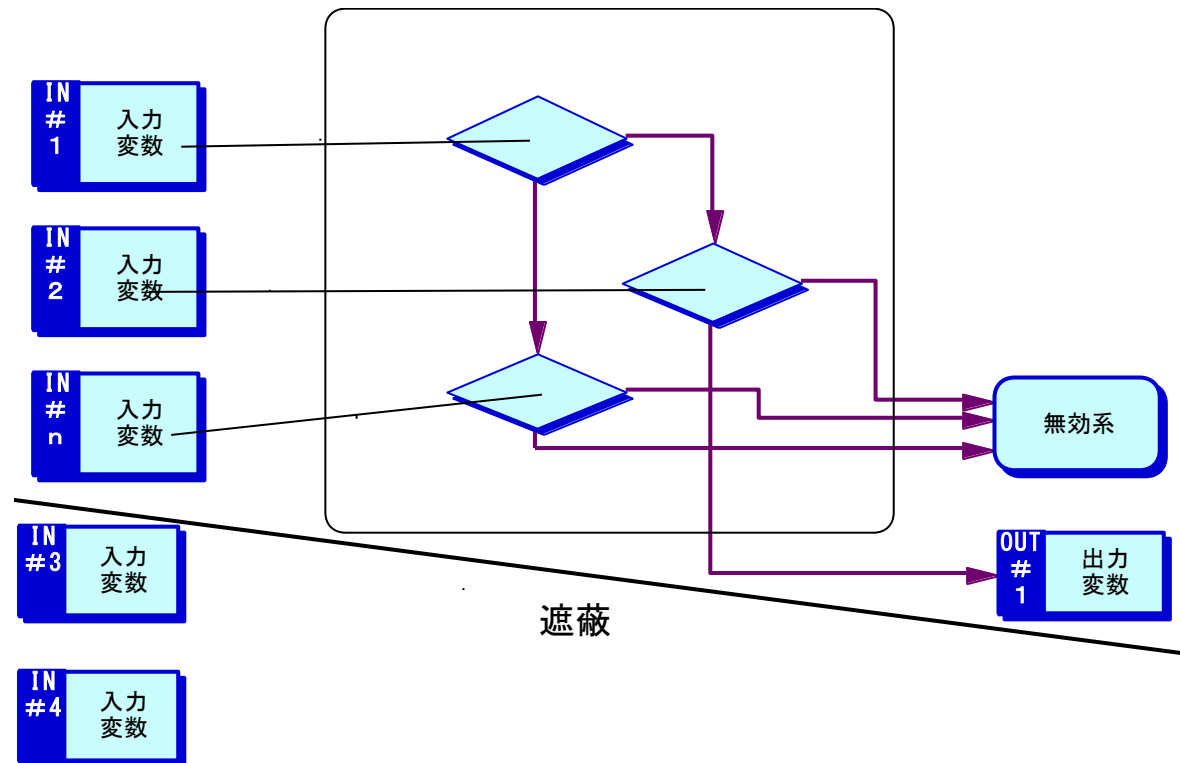


■ 構造仕様

- モジュールの結合
- クラス図

■ 原因の参照関係

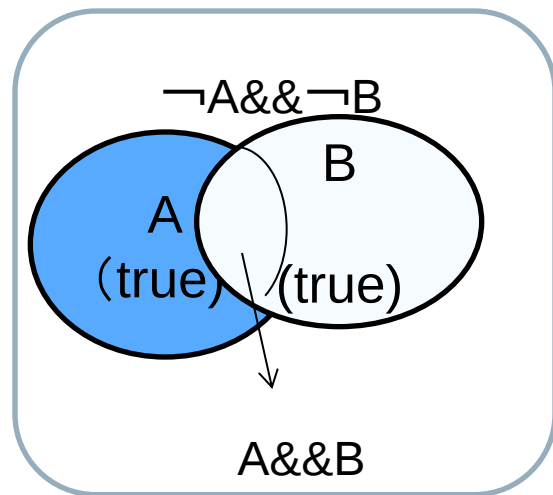
- 処理の流れと対応
- 無効系の追加



- 単体テストと結合テストの分離
- 単体テストでは網羅（仕様と実装）
 - 仕様のテストは同値分割（完全同値分割）と条件組合せ
 - 組合せのために：マトリックス、デシジョンテーブル（詳細）
 - 実装のテストは仕様のテストでは通過しないパス
- 結合テストでは効率化
 - 構造情報の利用：プログラム構造のブロック化が前提
 - 仕様実現の処理の流れを把握して図示
 - デシジョンテーブル展開（粒度粗）
 - 有効系の優先・・・無効系を減らす（組合せる必要のない因子）
 - 階層化・・・横連結方式の結合テストを実施。
- 基礎知識
 - 同値分割、境界値分析、マトリックス
 - デシジョンテーブル

距離割引、学生割引の組合せ

ベン図

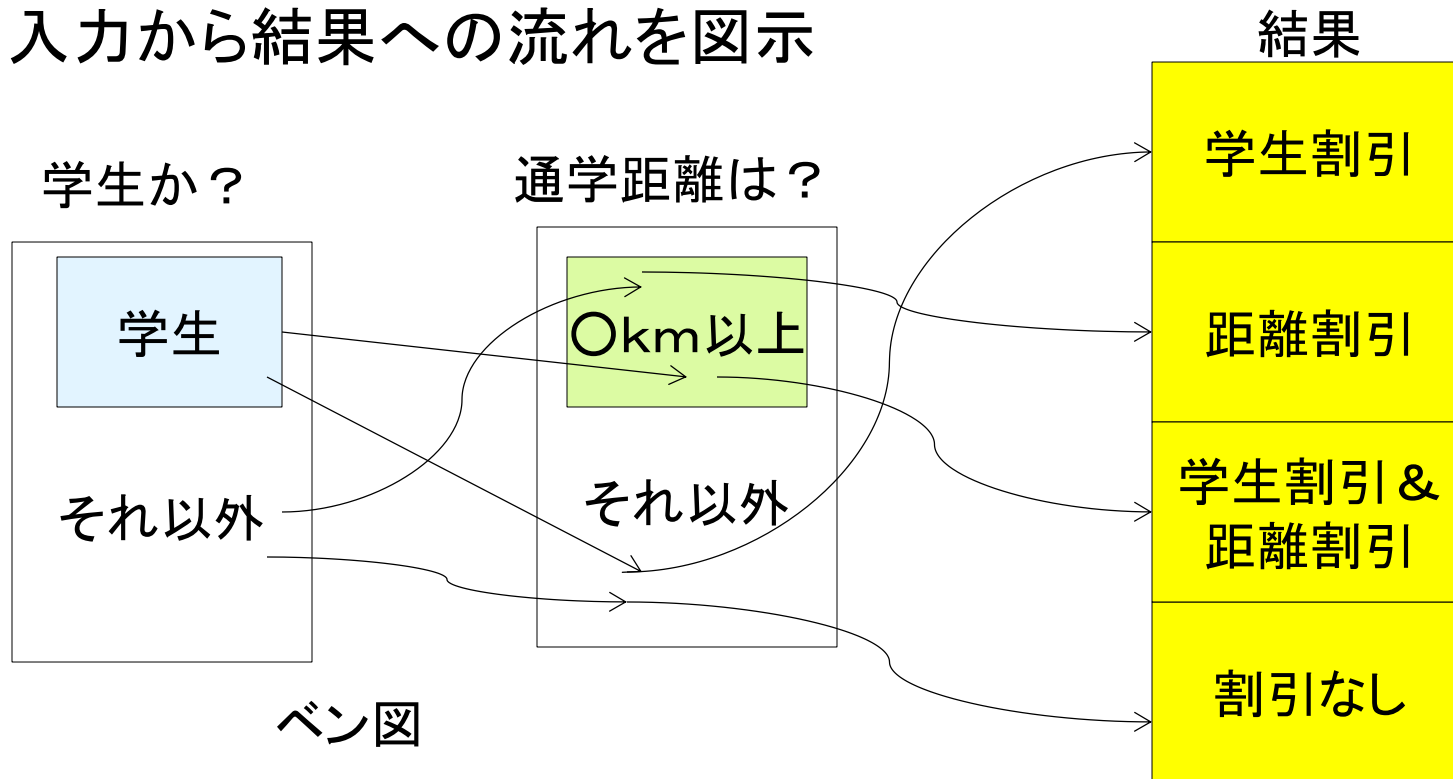


空間の各領域は
同値原因の集合

A(学生割引)	B(距離割引)	評価結果(割引)
false	false	なし
true	false	学割
false	true	距離割
true	true	学割・距離割

ベン図により、難解な集合概念を容易に理解できる

- 処理条件(学割、距離割)の同値分割
- 結果の同値分割、
- 入力から結果への流れを図示

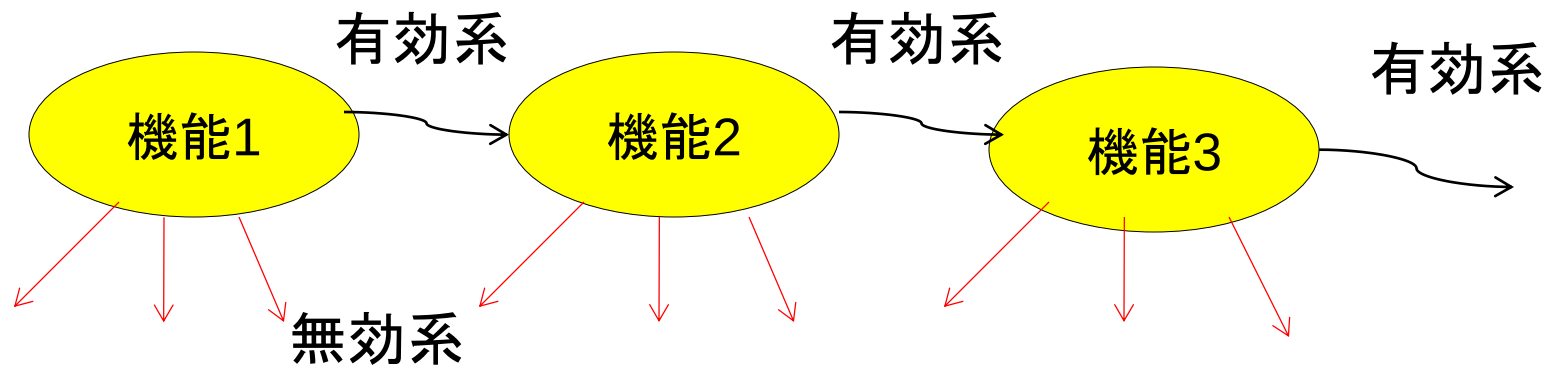


- 単体テストでは詳細な同値分割(条件ごと)
- 結合テストでは粗い同値分割(ブロックの処理結果)

■ プログラムの機能

- 有効系: 機能が働く 有則
- 無効系: 機能が働かない、エラー 無則

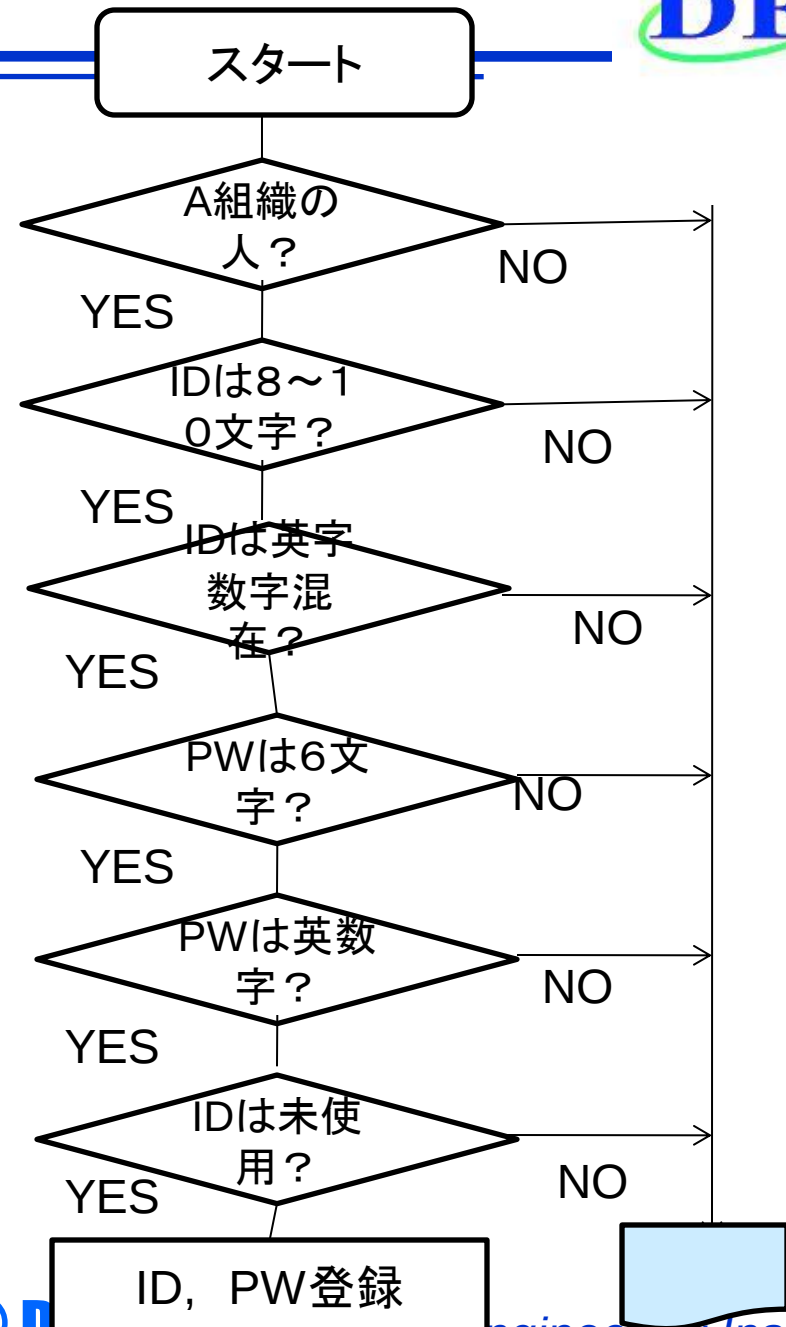
無効系は途中で処理が終了することが多い。



- 無効系は下に落ちる。無効系間の組み合わせはない。
- 結合テストでは有効系間の組み合わせを重視する。

組み合わせる必要のないケースの存在

- 社内のA組織の人のみ、IDとPWを登録できる。
- IDは8文字～10文字の英数字が混じった未使用のもの、
- PWは6文字の英数字でなければならない。

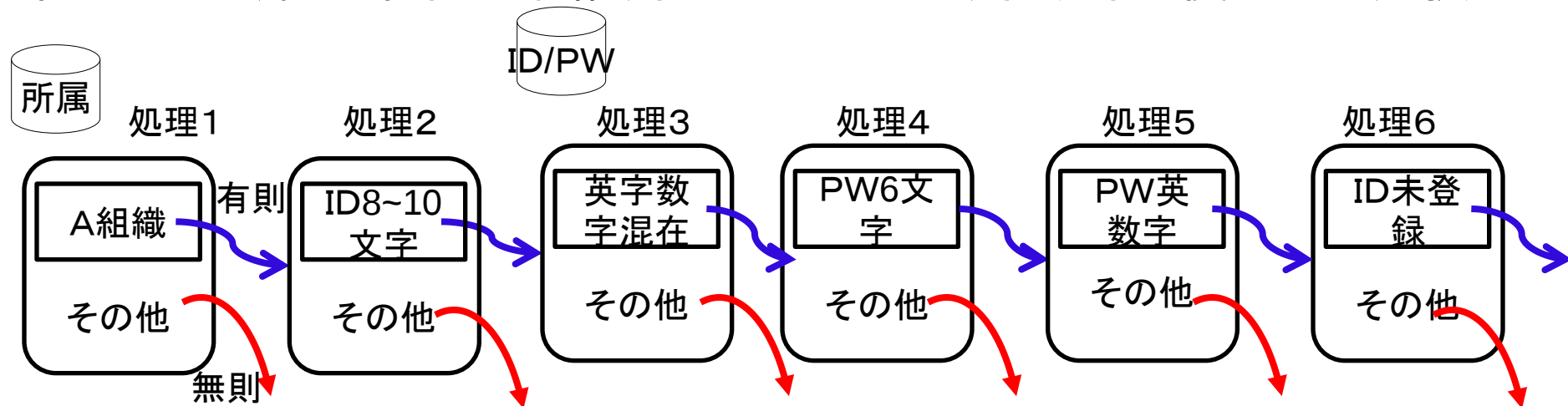


◆ 単純にマトリックスにすると 64ケース

			ID 8～10文字				ID 8～10文字以外			
			英数字混在		英数字混在以外		英数字混在		英数字混在以外	
			未使用	使用	未使用	使用	未使用	使用	未使用	使用
PW 6文字	PW 英数字	A組織	登録	NG	NG	NG	NG	NG	NG	NG
		他組織	NG	NG	NG	NG	NG	NG	NG	NG
	英数字以外	A組織	NG	NG	NG	NG	NG	NG	NG	NG
		他組織	NG	NG	NG	NG	NG	NG	NG	NG
PW 6文字以外	PW 英数字	A組織	NG	NG	NG	NG	NG	NG	NG	NG
		他組織	NG	NG	NG	NG	NG	NG	NG	NG
	英数字以外	A組織	NG	NG	NG	NG	NG	NG	NG	NG
		他組織	NG	NG	NG	NG	NG	NG	NG	NG

制御構造が分かっている場合テストケースは7

単体テストで、論理検証を網羅、結合テストでは、有効系を優先的に選択

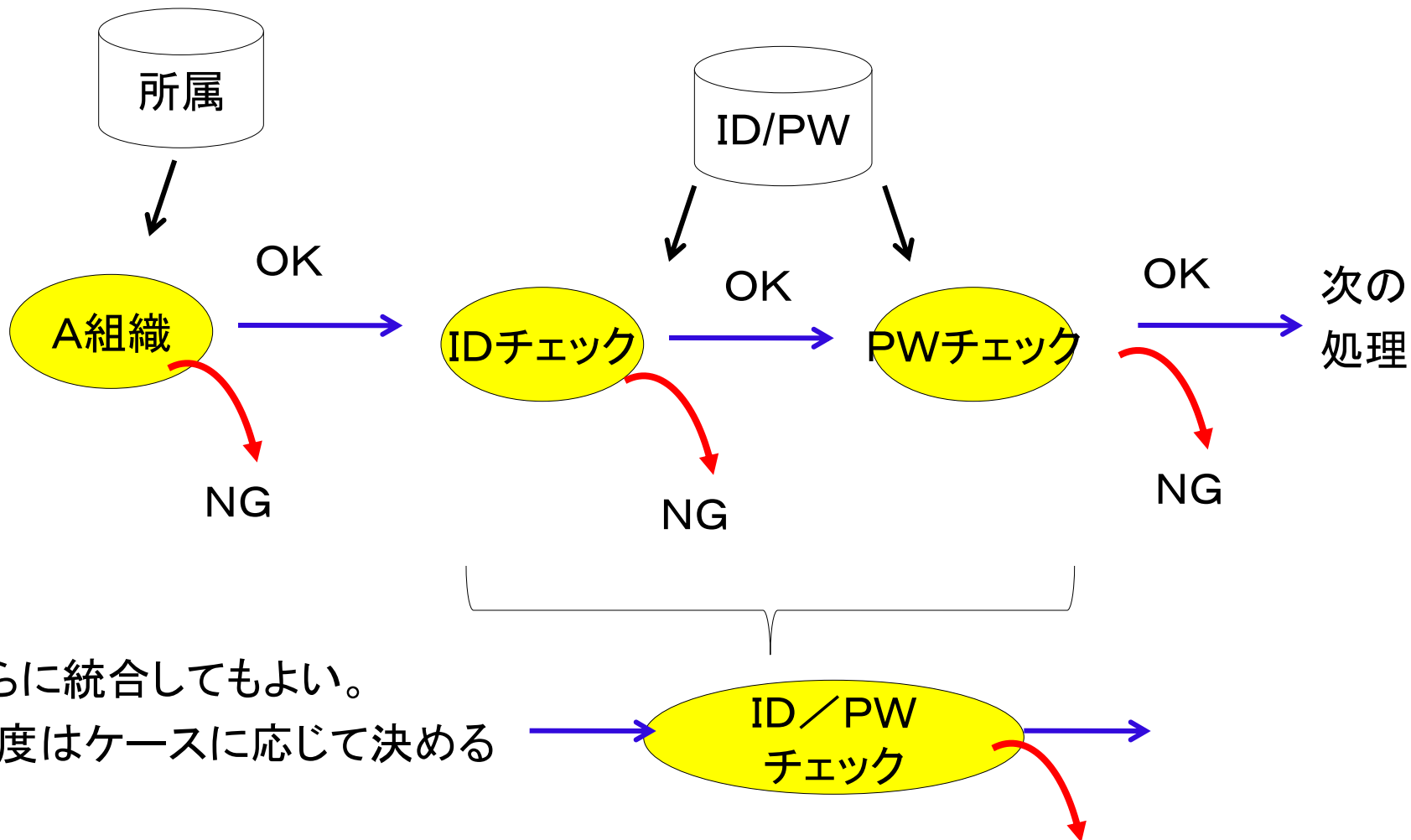


テストケース		①	②	③	④	⑤	⑥	⑦
原因	A組織所属	Y	N	Y	Y	Y	Y	Y
	ID 8~10文字	Y		N	Y	Y	Y	Y
	英数字混在	Y			N	Y	Y	Y
	PW6文字	Y				N	Y	Y
	PW英数字	Y					N	Y
	ID未使用	Y						N
結果	ID登録	Y						
	エラーMSG		Y	Y	Y	Y	Y	Y

制御構造を活かしたデシジョンテーブルを作成して、テストケースを設計する

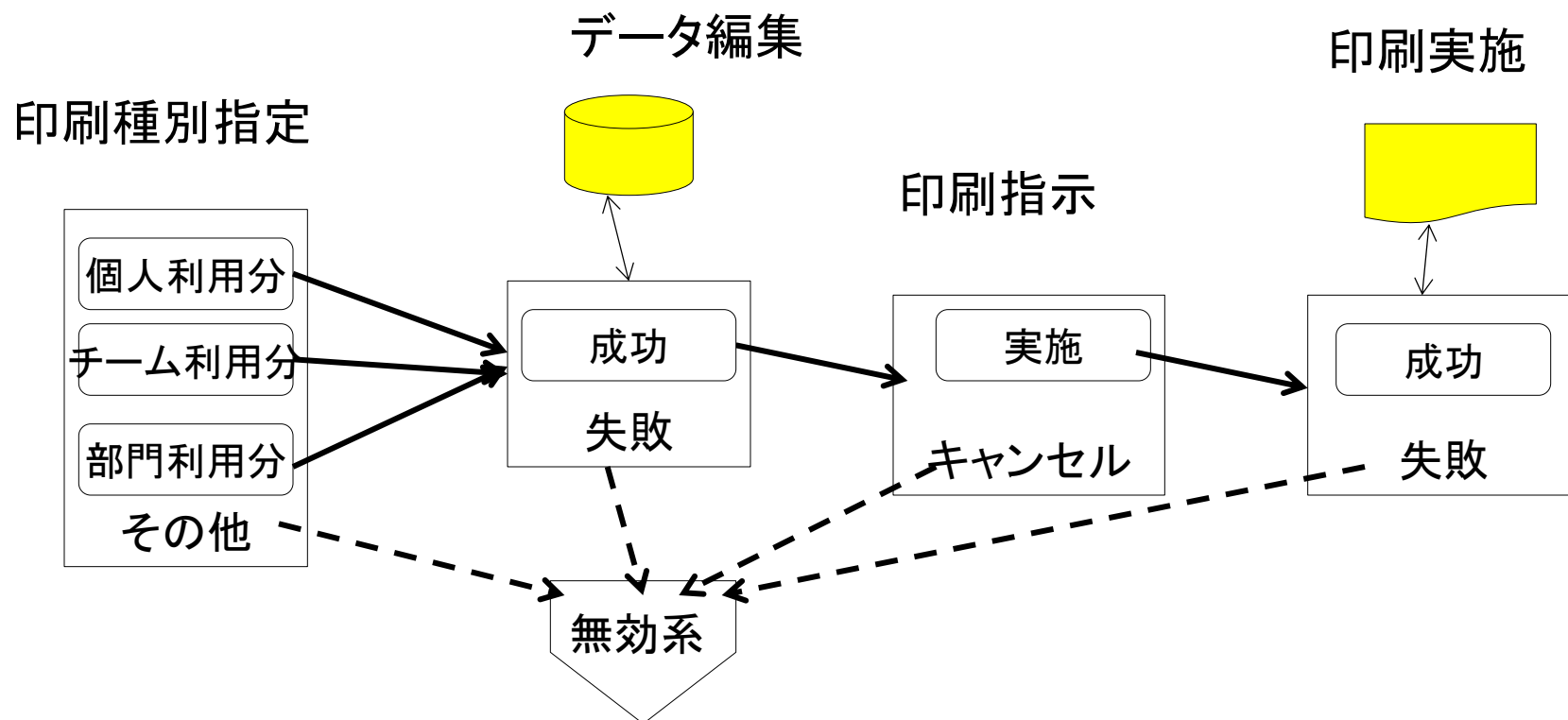
パスが長いケースでは粒度を粗くする

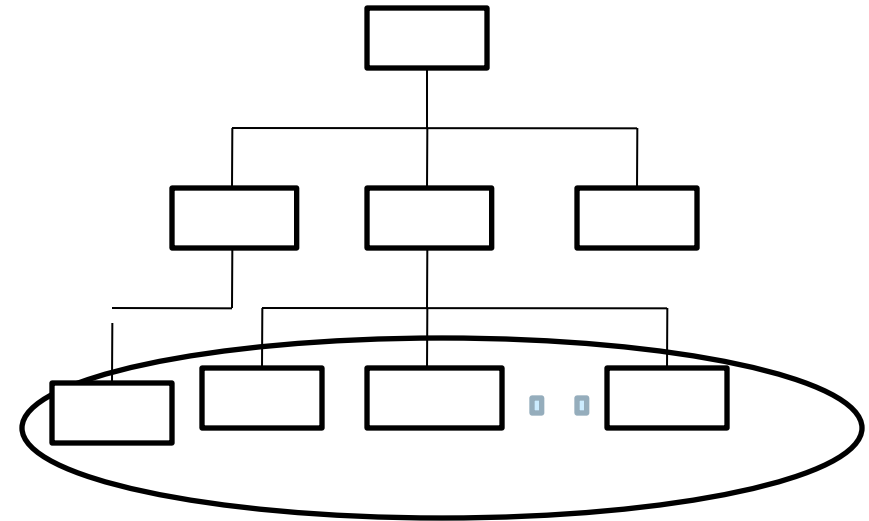
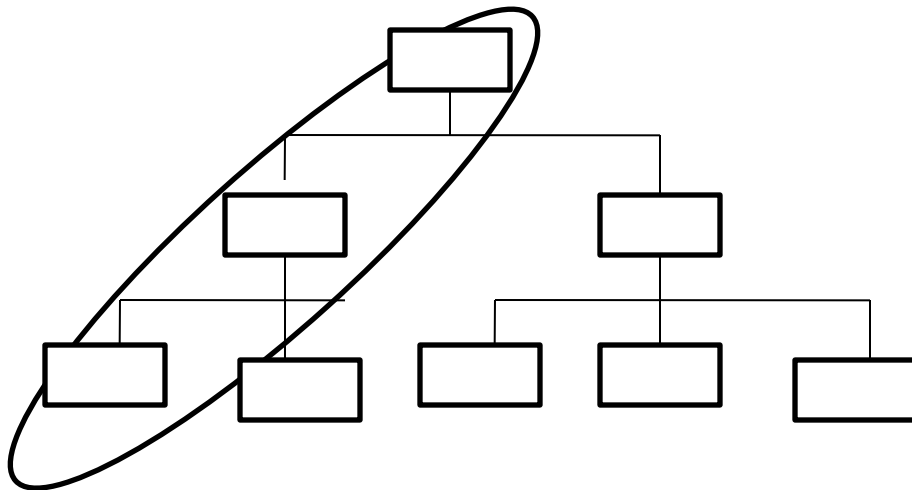
■ より簡単な図にする



■ 無効系は削減

例えば、パラメータで複数機能を指定できる場合、その処理過程で無効となるケース(キャンセル、異常等)のテストは1種類に絞る(個人利用、チーム利用、部門利用のどれか)





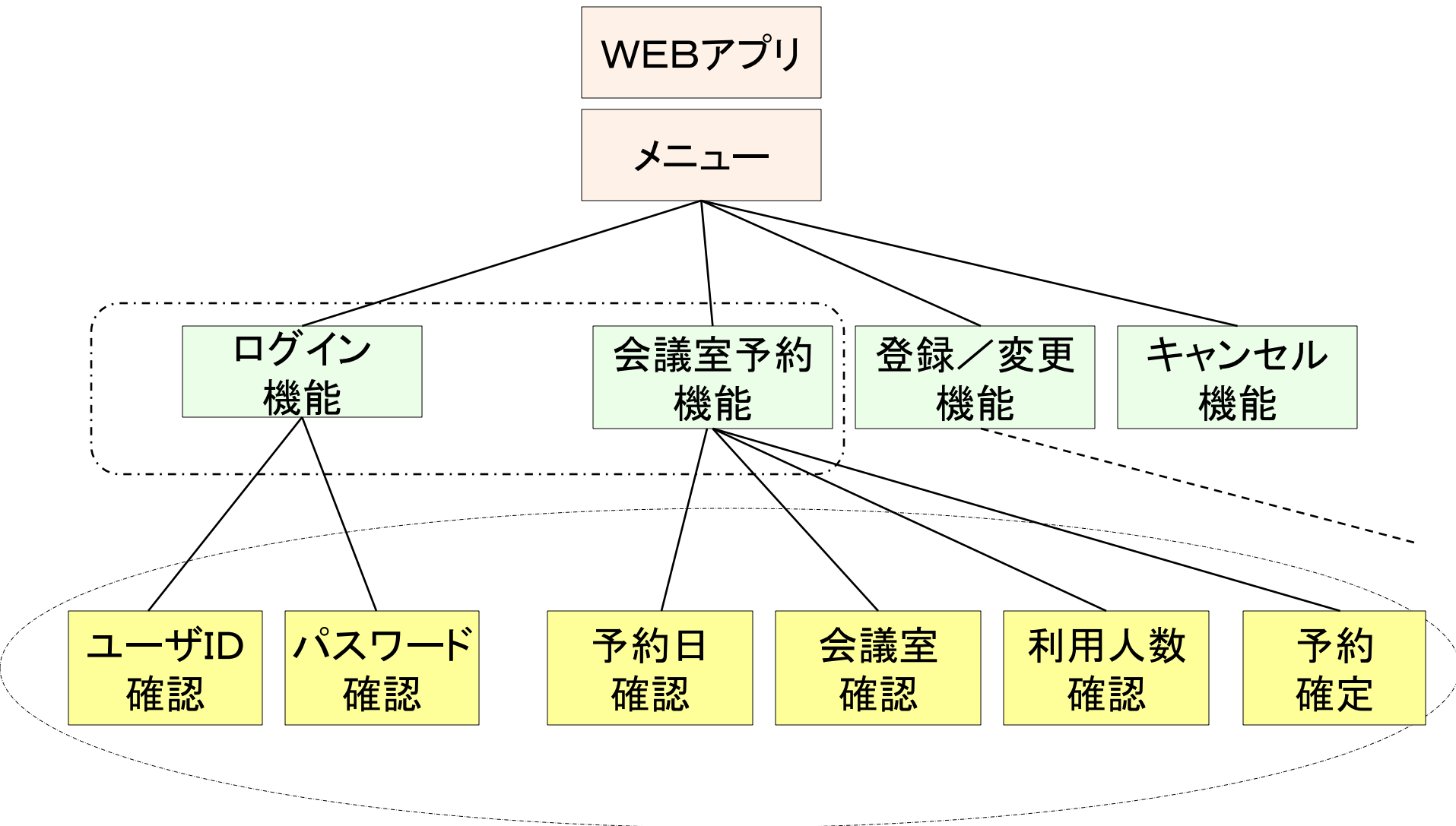
縦型:独立した機能間で分割 ×

- 分割は簡単だが
結合範囲が狭くなる
- 分割が細かくなると単体テストと変わらない
- 分割が大きくなるとテストケースが膨大になる

横型:階層で分割 ○

- 機能を階層で分割する
- 縦割りより難しくなるが、漏れなく分割できる
- テストケースの数は少なくできる

■ 階層分けの概念



- テストを効率的に実施するためには構造化
- 仕様に潜む論理を見つけ出し、それを見える化
- プログラムの論理構造は次で成り立っている
 - 機能とそれを動作させる条件(組合せ)
 - 条件間の関係性(排他、独立)
 - 機能間の関係性(排他、並列)
 - 処理の順番(エラーチェックの順番・位置、優先処理・・)
 - 処理の分割可能性(長い処理は分割、中間結果を使って繋げる)
 - その他
- これらの関係を表現
 - CFDでは決定表を使って見える化する

2. CFD++ ～CFDの進化の方向～

- 論理設計段階でテストし易さを確保する
- テスト技術の進展とCFD

■ テストでは仕様どおりの処理ができているか検証

■ 仕様とは

1. 目的機能としての仕様: ビジネスルール, 制御ルールなど
2. 実装機能としての仕様: 開発過程における仕様

■ CFDは, 1+2の仕様を明らかにする手法

- 流れ図を使って, 制御論理を決定表で表現する

■ CFD適用上の課題

- 適切な論理構造設計と論理構造の明示・表現が前提

■ この解決法として, <設計段階での制約>

- それを実現する, 設計段階で使うCFD++ (検証指向型のCFD++)

■ 加えて、近年、静的テスト(静的解析)の進歩から、

- テストデータ作成, 実行などを省略する画期的なツールが出現
- ツールを活用し, テストを省力化するCFD++へ

2. 1 設計段階での制約

2. 1 設計段階での制約

- 現在の構造設計はモジュール構造(関係)の表現中心
- 論理構造の表現手段の必要性 → 決定表
 - 単適合、多重適合決定表の使い分けと組合せ
 - 論理構造と表現方法を対応させる
 - 仕様の持つ論理構造を読み取って表現

2. 1. 1 単適合決定表 JIS X0125

ビジネスルールを仕様化する場合,

■ 個々の判定と, 判定間の関係の両方を定義する

■ 次の仕様: 個々の判定は,

- 老齢割引の対象は, 女性の場合には50歳以上, 男性の場合には60歳以上とする.
- 3歳以下の場合は, 幼児割引(無料)とする.
- 12歳以下の場合は, 年少割引とする.
- 年齢が120歳以上は, エラーとする.
- 水曜日に限り, 女性はレディ割引を行う.
 - 但し, 老齢割引, 年少割引, 幼児割引対象者は除く

■ 判定間の関係

- 幼児割引と年少割引
- {老齢割引, 幼児割引, 年少割引}とレディ割引
- エラーと他の判定

判定間の関係を定義する

1. 単適合型 (single-hit) の決定表

- ある範囲内において、判定は一つのみ実行される。
- 複数の判定が成立可能性があっても、優先順位によって判定は一つ
 - ビジネスルールとして、多く見られるタイプ

2. 多重適合型 (multiple-hit) の決定表

- 成立した判定は、独立にそれぞれが実行される

■ ビジネスルールの仕様化において重要な概念

- 例 3歳以下の場合は、幼児割引(無料)とする。
- 12歳以下の場合は、年少割引とする。
- 以上は、単適合型の表現
- 多重適合型なら、4歳以上12歳以下の場合は、年少割引とする。
 - プログラミング段階で、両方の割引が実行されるコードを書く可能性あるので、範囲を明確に示す必要が生ずる。

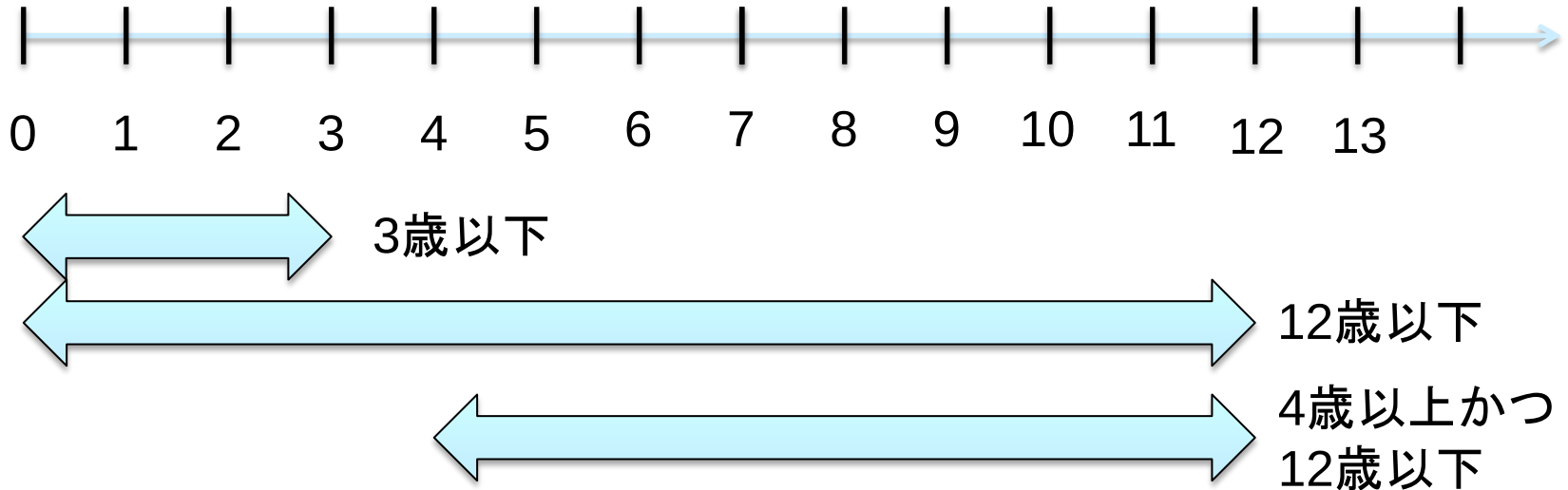
- 例： 割引問題のように、似たようなサービス（割引）があり、同時には適合（hit）しないビジネスルール.
- 単適合を表現する決定表：単適合決定表

規則（単適合）		1	2	3	4	5	6
条件部	年齢	50歳以上	60歳以上	3歳以下	12歳以下	120歳以上	
	性別	女性	男性				女性
	曜日						水曜日
動作部	老齢割引	X	X				
	幼児割引			X			
	年少割引				X		
	エラー					X	
	レディ割引						X

- 規則の優先順位
1番から順に評価，成立すれば以降の規則を評価しない
- 条件部
拡張指定と制限指定
表は拡張の例
制限では条件部に詳細
ブランクはANY
- 動作部
X：動作する
ブランク：動作しない

■ 包含関係

- 「3歳以下」は「12歳以下」に含まれる



- 仕様: 3歳以下なら, 幼児割引(無料)が使える.
 - 12歳以下なら, 年少割引が使える.
- ☆ 3歳児は, 幼児割引と年少割引の両方が使えると読める
一つしか使えない(排他的): 3歳児は年少割引でも良い?
単適合: 判定の順序まで指定する. <誤解がなくなる>
あるいは, 年少割引は, 4歳以上かつ12歳以下とする.

- たとえ、判定(規則)間に包含関係があったとしても、そのことを考慮せずに1番から順に評価し、適合(hit)すれば、以降の判定(規則)は評価しない。

規則(単適合)		1	2	3	4	5	6
条件部	年齢	50歳以上	60歳以上	3歳以下	12歳以下	120歳以上	
	性別	女性	男性				女性
	曜日						水曜日
動作部	老齢割引	X	X				
	幼児割引			X			
	年少割引				X		
	エラー					X	
	レディ割引						X

6
13歳以上 49歳以下
女性
水曜日
X

規則の重複を防ぐ記述

2. 1. 2 多重適合決定表

単適合では手間が増える場合

■ ある割引仕様

- 住民割引: 区内に住民票を持つ場合には, 他の割引とは別に15%割引
- ネット割引: ネットで購入するとさらに10%割引
- 夜間割引: 18時以降の場合, さらに10%割引
- 割引の併用は, 割引率の加算による. 例 10%と15%なら, 25%割引

■ この仕様の判定や動作は, お互いが独立している

多重適合: それぞれの動作は独立している

■ 単適合で表現すると, 全組合せを考える必要がある

- 順位1 3つの割引が同時 1ケース
- 順位2 2つの割引が同時 3ケース
- 順位3 1つの割引のみ 3ケース

単適合決定表で記述すると

- 起こりえる, すべての組合せ8つを洗い出し
- 動作が何も無い1つを除く7つについて定義する
- 順序については, 組合せ数の多いものから順に記述する……大変

規則(単適合)		1	2	3	4	5	6	7
条件部	住民か	Y	Y	Y		Y		
	ネット購入か	Y	Y		Y		Y	
	18時以降か	Y		Y	Y			Y
動作部	住民割引加算	X	X	X		X		
	ネット割加算	X	X		X		X	
	夜間割加算	X		X	X			X

- 単適合決定表(JIS)の欠点を補完する: 多重適合
- 判定(規則)間の関係は考えず(独立として)決定表を記入する

規則(多重適合)		1	2	3
条件部	住民か	Y		
	ネット購入か		Y	
	18時以降か			Y
動作部	住民割引加算	X		
	ネット割加算		X	
	夜間割加算			X

- 多重適合の明示

- 規則はすべて評価

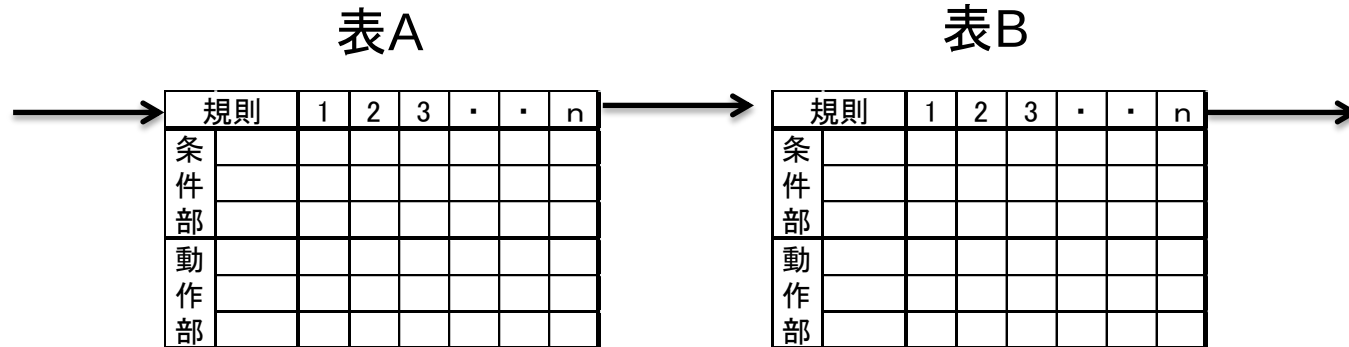
1番から順に最後まで評価し, それぞれの規則(判定)に従って, 多重に処理を行う.

2. 1. 3 決定表の分割と結合

結合の入口、出口

■ デフォルトとして:

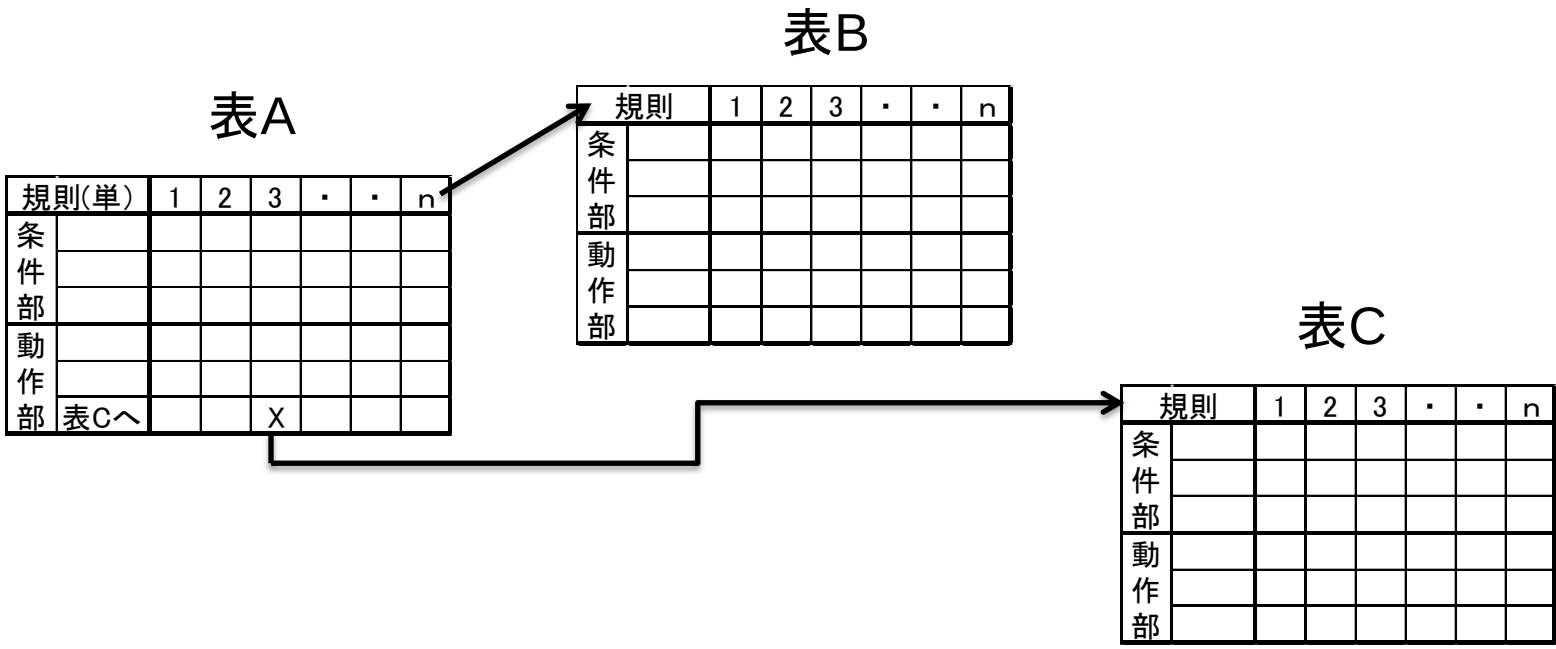
- 単適合, 多重適合の両方とも入り口は規則
- 出口は, 規則の終了



■ 仕様書において, 特に明示しない場合はこの結合

- 決定表は, 順次評価される

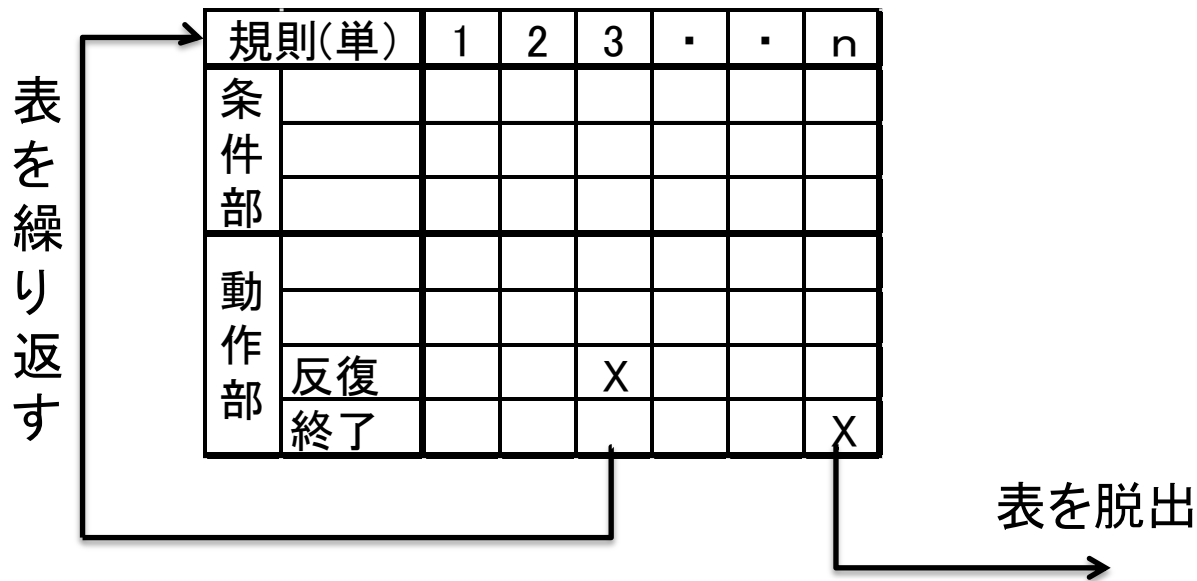
- 主に単適合において、動作部に他表への分岐を記入



- 表Aの規則#3が成立し、指定された動作の最後に表Cへ飛ぶ.
 - 表Cにおいて、表Aへ戻る指定も可能(サブルーチン的な使用)

■ 表を繰り返す

- 反復動作を含む規則が成立すると、反復
 - － 規則3が成立するとこの表を繰り返す
- 終了についても記載する必要がある
 - － 規則nが成立するとこの表を終了する



特別な規則: ELSE

■ ELSE規則

- 無条件に成立する規則
- 規則の最後にのみ記載する(途中にあると、以降の規則は常に打ち切り)

■ 単適合の場合のみ

- ELSE規則より左側にある規則が一つも適合しない場合にのみ適合する.

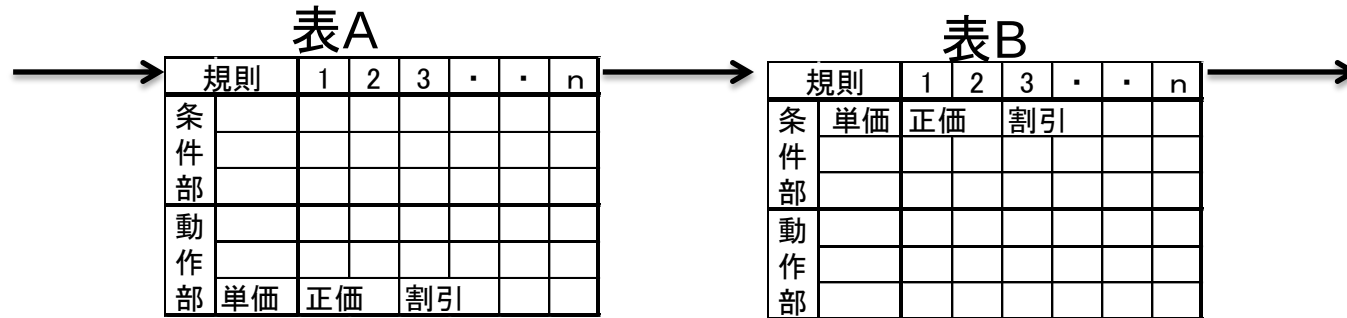
■ 多重適合の場合

- ELSE規則は使えない.

規則		1	2	3	・	・	n
条件部							ELSE
動作部							
							X

■ 論理の分割に伴う繋ぎ

- 表Aの出力(中間出力)を
- 表Bの入力とすることもできる



■ 中間出力とは

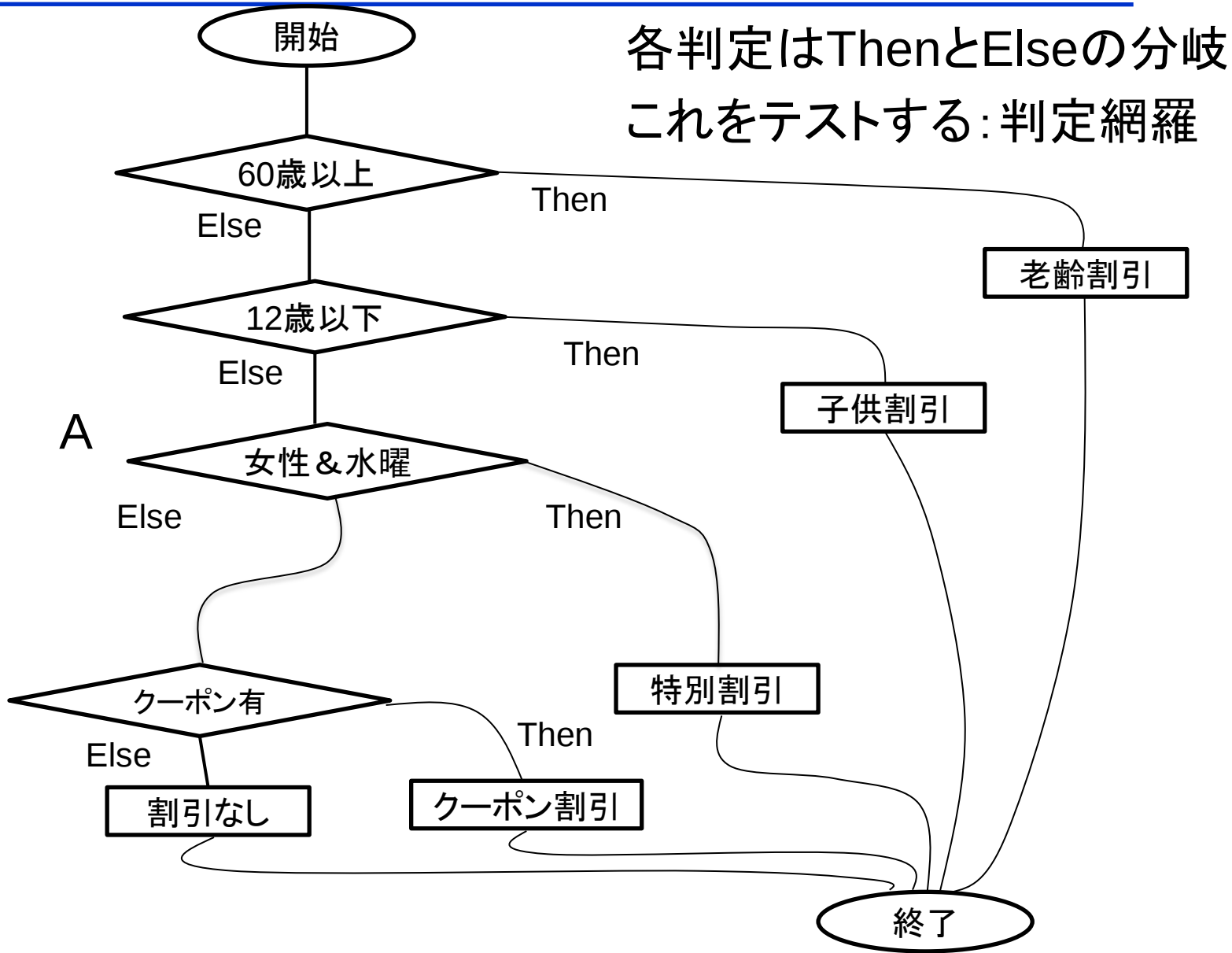
- 論理の複雑化を防ぐ工夫
- 検証を容易化
- 計算方法とデータなど

2. 1. 4 テストの決定表

■ 単適合決定表の場合

- 「60歳以上なら老齢割引(20%), 12歳以下なら子供割引(40%), 女性で水曜日に限り特別割引(15%), 割引券持参ならクーポン割引(15%)をそれぞれ行う。割引は加算されない。」
- 仕様の決定表は以下

規則(単適合)		1	2	3	4
条件	年齢	60以上	12以下		
	性別			女性	
	曜日			水曜	
	割引券				持参
動作	老齢割引	X			
	子供割引		X		
	特別割引			X	
	クーポン割引				X



テストの決定表: 判定網羅基準を踏まえて

■ 増えるのは、結果がすべて「ELSE」のケース

¬(年齢 ≥ 60) & ¬(年齢 ≤ 12) & ¬(性別 = 女性 & 曜日 = 水曜) & ¬(割引券 = 持参) このケースを追加する。

テストケース			1	2	3	4	5
原因	年齢	60以上	Y				
		12以下		Y			
		その他			Y	Y	Y
	性別 = 女性		—	—	Y	N	Y
	曜日 = 水曜		—	—	Y	Y	N
	割引券 = 持参		—	—	—	Y	N
結果	老齢割引		X	N	N	N	N
	子供割引		N	X	N	N	N
	特別割引		N	N	X	N	N
	クーポン割引		N	N	N	X	N

多重適合決定表のテスト

■ 仕様:

住民であれば住民割引(10%)、ネット購入の場合はネット割引(5%)、18時以降の入場は夜間割引(10%)が加算される。同時加算可能。

■ 多重適合では規則(判定)間は独立

■ 1の規則の後、2の規則、そのあと3の規則を通る。

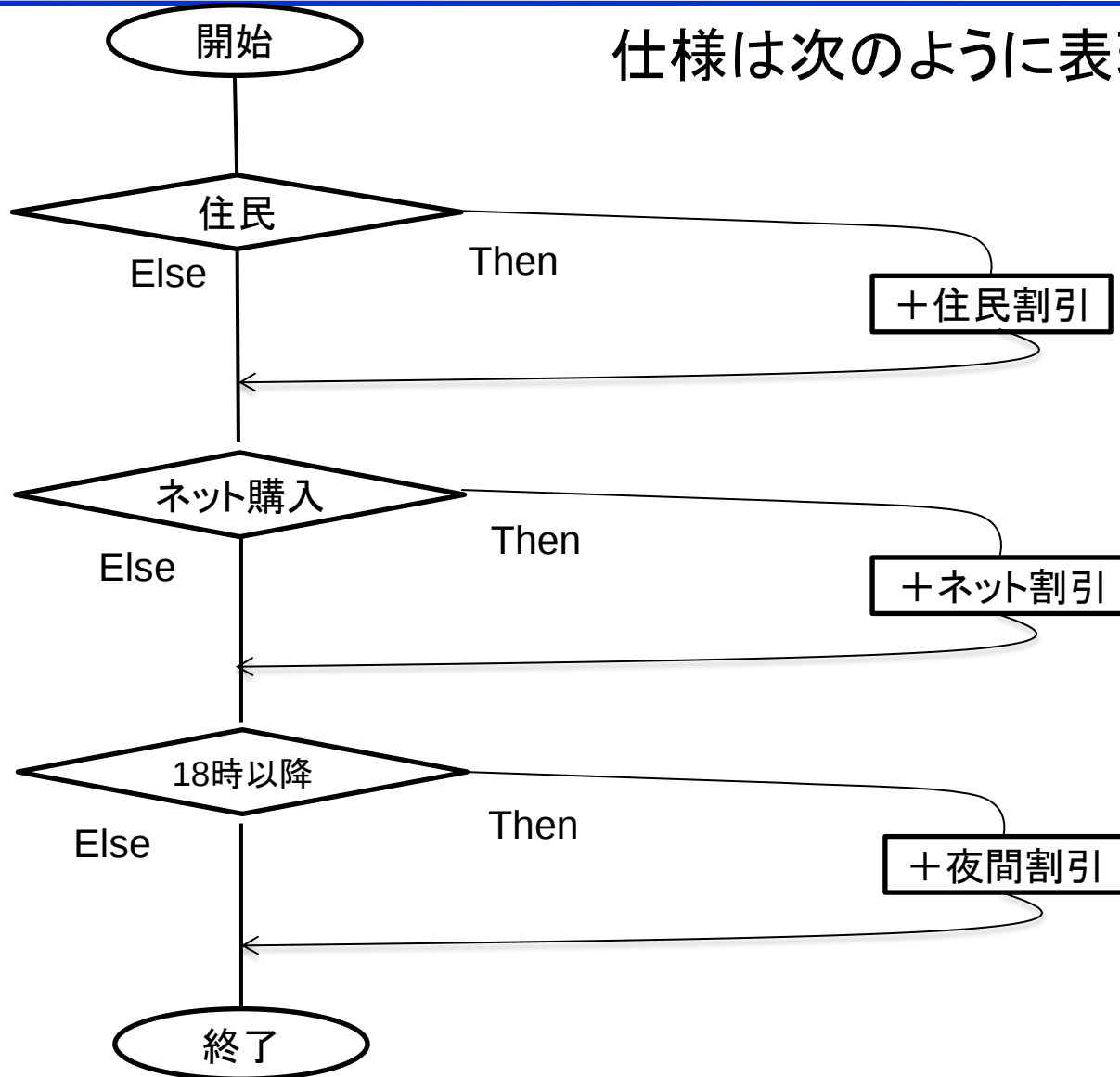
規則(多重適合)		1	2	3
条件部	住民か	Y		
	ネット購入か		Y	
	18時以降か			Y
動作部	住民割引加算	X		
	ネット割加算		X	
	夜間割加算			X

● 仕様の決定表

1番から順に最後まで評価し、それぞれの規則(判定)に従って、多重に処理を行う。

フロー図で考える: 多重適合

仕様は次のように表現できる。



テストの決定表: 多重適合の仕様決定表から

- 判定網羅基準では
- 各判定ごとのThenとELSEをテストする。
 - 住民割引: Then, Else の2ケース
 - ネット割引: Then, Else の2ケース
 - 夜間割引: Then, Else の2ケース
- テストでは統合する
 - テストの決定表は次。

テストケース		1	2
原因	住民	Y	N
	ネット購入	Y	N
	18時以降	Y	N
結果	住民割引	X	N
	ネット割引	X	N
	夜間割引	X	N

テストの決定表2

- 仕様
- 60歳以上なら老齢割引(20%), 12歳以下なら子供割引(40%), 女性で水曜日に限り特別割引(15%), 割引券持参ならクーポン割引(15%)をそれぞれ行う。これらの割引は加算されない。
- さらに、上記に以下が加算される。

住民であれば住民割引(10%)、ネット購入の場合はネット割引(10%)、18時以降の入場は夜間割引(10%)が加算される。同時加算可能。

仕様の連結決定表

規則(単適合)		1	2	3	4
条件	年齢	60以上	12以下		
	性別			女性	
	曜日			水曜	
	割引券				持参
動作	老齢割引	X			
	子供割引		X		
	特別割引			X	
	クーポン割引				X

規則(多重適合)		1	2	3
条件部	住民か	Y		
	ネット購入か		Y	
	18時以降か			Y
動作部	住民割引加算	X		
	ネット割加算		X	
	夜間割加算			X

テスト決定表の統合

テストケース			1	2	3	4	5
原因	年齢	60以上	Y				
		12以下		Y			
		その他			Y	Y	Y
	性別＝女性		－	－	Y	N	Y
	曜日＝水曜		－	－	Y	Y	N
	割引券＝持参		－	－	－	Y	N
結果	高齢割引		X	N	N	N	N
	子供割引		N	X	N	N	N
	特別割引		N	N	X	N	N
	クーポン割引		N	N	N	X	N

テストの決定表1のテストケース

テストケース		1	2
原因	住民	Y	N
	ネット購入	Y	N
	18時以降	Y	N
結果	住民割引	X	N
	ネット割引	X	N
	夜間割引	X	N

テストの決定表2のテストケース

- それぞれにテスト項目を作る
- 両者を組み合わせて統合する。

テスト決定表の統合

- 2つの決定表を組み合わせる
- テストケースの多い決定表に、少ない方を全て移す。(黄網部分)
- 残るテストケースには、適切な組合せをわり振る。(青部分)

テストケース			1	2	3	4	5
原因	年齢	60以上	Y				
		12以下		Y			
		その他			Y	Y	Y
	性別=女性		—	—	Y	N	Y
	曜日=水曜		—	—	Y	Y	N
	割引券=持参		—	—	—	Y	N
	住民		Y	Y	N	N	N
	ネット購入		Y	N	Y	N	N
	18時以降入場		Y	N	N	Y	N
結果	老齢割引		X	N	N	N	N
	子供割引		N	X	N	N	N
	特別割引		N	N	X	N	N
	クーポン割引		N	N	N	X	N
	住民割引		X	X	N	N	N
	ネット割引		X	N	X	N	N
	夜間割引		X	N	N	X	N

テスト決定表の展開にあたって

- 例は判定網羅を基準にした結合テスト
- 実務では必要に応じて基準を選択する
 - 例えば結果データの組合せ
 - 全割引率の種類など
- 単体テストでは別の網羅基準

- CFD++ その1 設計段階での制約とは
 - 構造設計結果を決定表で表現すること
 - そのためには、(仕様+実装)論理構造の分析
 - 論理構造の例として単適合型、多重適合型
 - ループなどは動作部の記述方法で表現可能
 - CFDの各ブロックを決定表に置き換えただけ?
 - ⇒ それ以上の意味

- *ConcollicTesting*: 静的解析を利用したテストケース設計、テストの実施
- *ConcreteTest*、*SymbollicTest*を使い分ける構造ベーステスト方法
- 問題は処理結果(出力)の正解値、仕様との照合
- *CFD++*の試み
 - 静的解析から決定表を自動生成、仕様から作成した決定表との照合

■ テーマ(目的)

- 単体テストに変革が生じている
- Concolic testingによるDynamic Test Generationの技術
- 利用可能になっているので, 実例で紹介

■ 概要, 構成

1. 自動化の背景: 静的解析の進歩
2. CUTE の出現
3. まとめ

2.2.1 自動化の背景: 静的解析の進歩

■ レビューから始まった静的テスト

■ レビュー: 古くから実践

- *Glenford J. Myers: The Art of Software Testing*
 - 1版 1979年 2版 2004年 3版 2011年
- *Chapter 3: Inspection, Walkthroughs, and Reviews*
 - 人間によるテスト技法, チームメンバーがプログラムを読みバグを発見する.

■ レビュー観点

- 動的テストの「仕様ベース, 構造ベース, 経験ベース」とほぼ同じ

■ 現在でも多用されている

- IPA調査によると, 70%の現場で行われている <品質屋さんは熱心>
- その効果は?

■ レビュー: 工学的サポートが希薄

- プロセス, 手順については文書化されている.
- 属人性に左右される.

■ 動的テストでは発見が困難なバグ

- メモリーリークなど, プログラミングにおける副作用として生ずるバグ
- 機種互換が保てないプログラムなど
- 膨大なテストを行っても見つかる確率がとても低い
- 原因を分析すると, 共通したプログラミングパターンが存在する
- 業界として, これを取りまとめ標準とした
 - C言語を安全に使用するためのプログラミング手引き
- “Motor Industry Software Reliability Association”がまとめた 1998 年

■ ツール化

- MISRA 手引きに合致していない(逸脱した)コードを見つけるツール
- ツール市場のニーズから, 静的解析が活気づいた
- 初期のツールは, ソースコードを探索するだけ
- その後, プログラム構造を基に探索するツールへ

■ 新技術とは静的解析によるテストデータ自動生成

- どんな原理なのか
- 何ができるのか？できないのか？
- どのように応用するのか？

■ 特に、現在のテストがどのように変わるのか？

- まだ、まだ始まったばかりです.

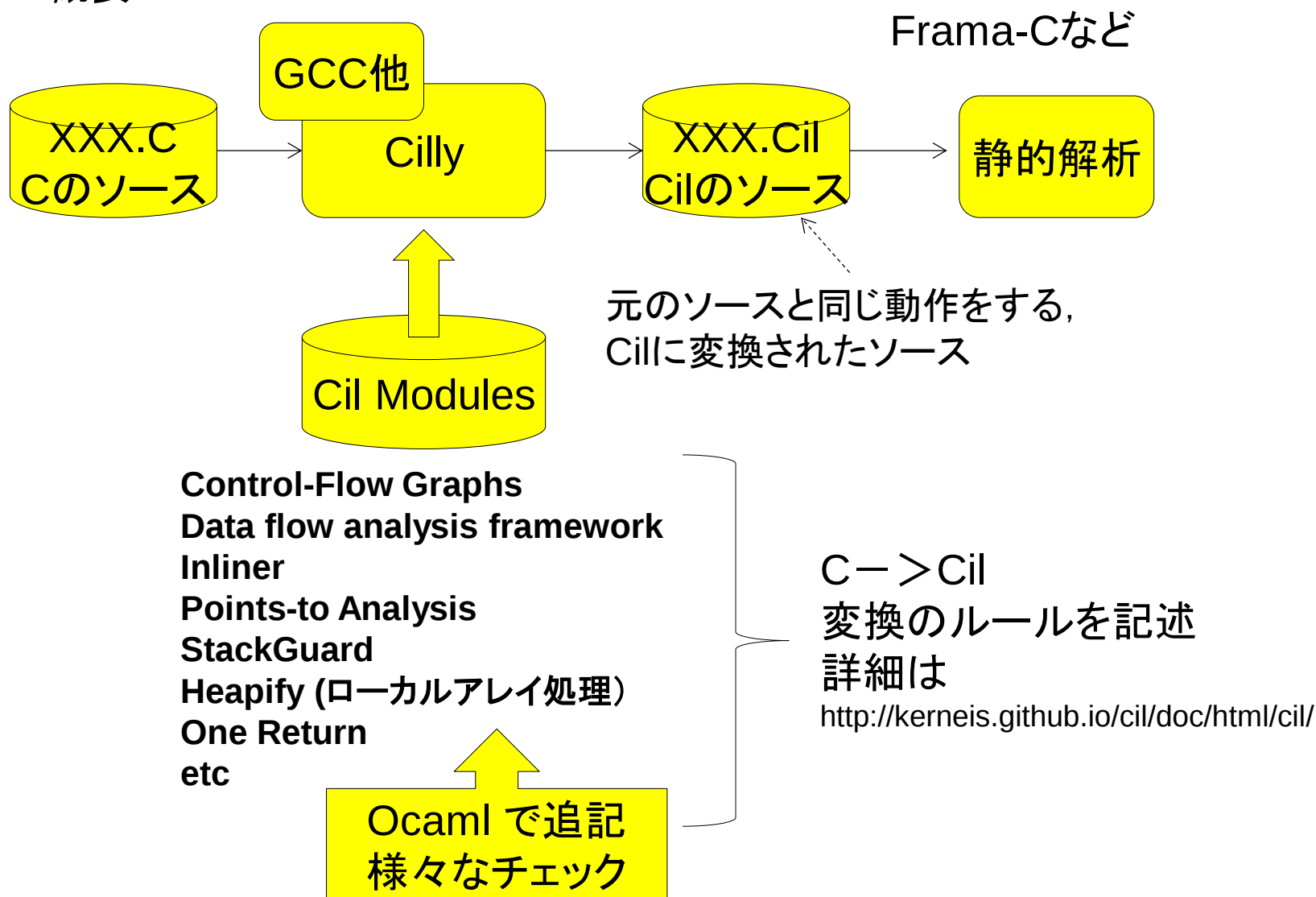
■ 静的解析のための前処理

- C言語を始め, プログラミング言語は曖昧な表現を含んでいる.
 - 機種互換や最適化において問題
- そこで, 厳密な表現に変換する技術
- CILは, 制御構造を解析し, 抽象構文木 (abstract syntax tree) で表現することにより曖昧性を排除する.
 - George C. Necula, etc ; CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs , CC '02 Proceedings of the 11th International Conference on Compiler Construction, pp 213-228
- バークレー大の研究で, オープンソースとして公開されている
 - コンパイラにgccを使い, 解析部分はOcamlで記述されている
- Frama-Cなど, プログラムの解析ツールは, CILを使って実装されている

■ CILにより静的解析の研究が加速された

- <http://kerneis.github.io/cil/>
- 別CIL: Common Intermediate Language(共通中間言語: Microsoft の.NET)

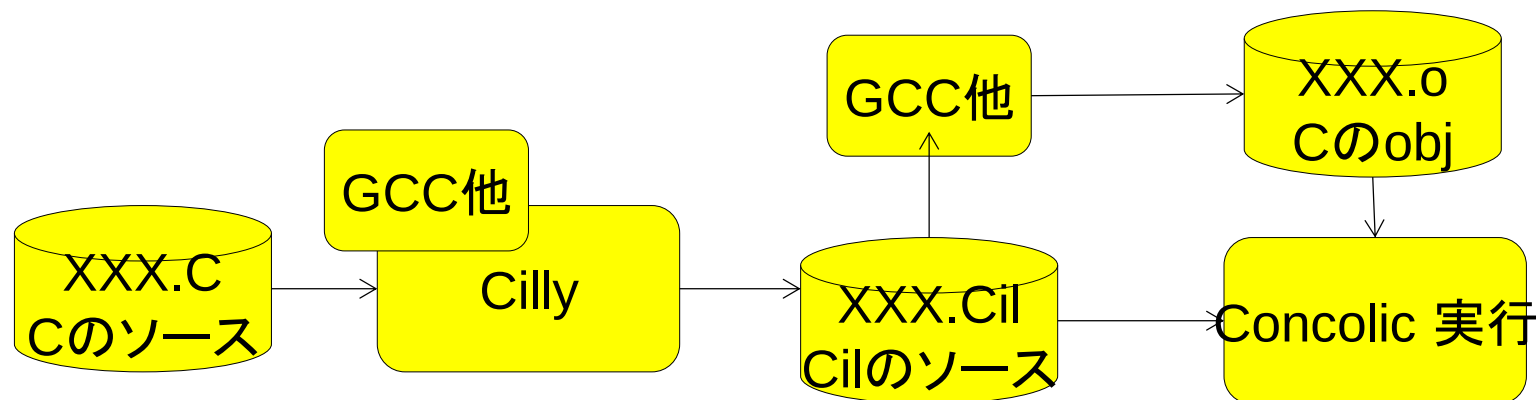
● 概要



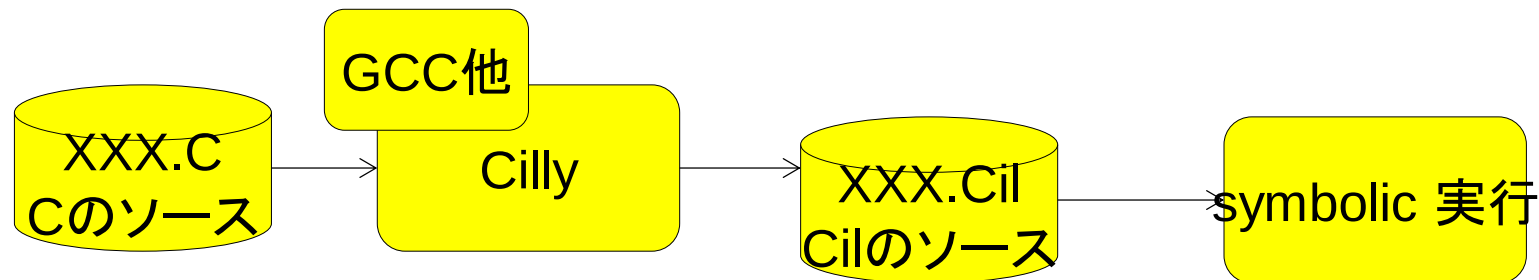
- CUTE: *concolic unit testing engine*
- この中で, *Concolic testing* なる用語が使われた
 - Sen, Koushik; Darko Marinov, Gul Agha (2005). "CUTE: a concolic unit testing engine for C". Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. New York, NY: ACM. pp. 263–272. ISBN 1-59593-014-0. Retrieved 2009-11-09.
- 機能: 実装されコードのコードカバレッジを最大にする, 具体的な入力変数の値(テストデータ)を生成する.
- 方式: 論理制約をソルバーを使って解き, 部分的な動的解析によりコードカバレッジを最大化する.
 - 制御フローの論理制約をソルバーを使って解く方法は, 以前からあった.
 - Path Crawler: (<http://pathcrawler-online.com>)
 - Frama-CのPathCrawler プラグイン

- ソルバーで解くために
 - 網羅すべき制御が実行されることを確認する
 - そのために、シンボリック実行が必要
- *symbolic execution* の改善
 - C言語の動作を完全にインタプリタするのはとても困難
 - そこで、Cillyによるフロント処理済み(AST化)の中間言語(Cil)を対象とする
 - さらに、一部の実行はpiggy-back方式で行う
 - concrete (execution) and symbolic execution = Concolic
 - Concrete execution : 実際の実行 (symbolic execution ではない普通の実行)
 - Concolicとは、インタプリタの一部を実実行で行うシンボリック実行処理系
- メリット:
 - インタプリタの実装が容易
 - 正確でかつ高速である

- Concolic Execution



- Symbolic Execution



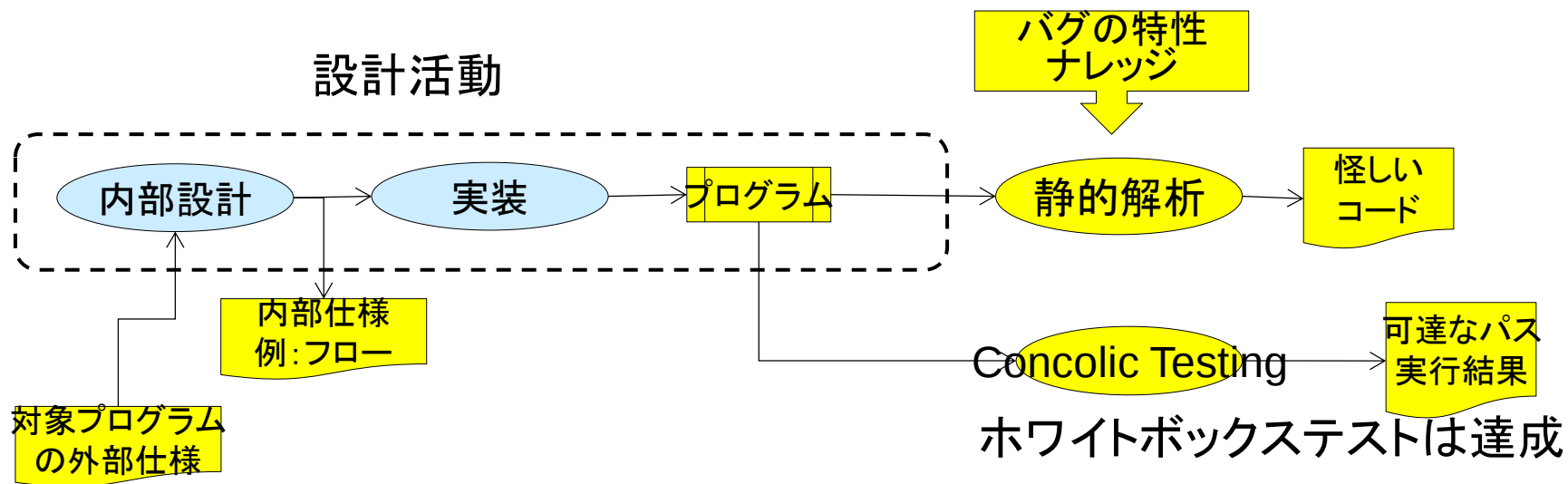
- *PathCrawler*: (<http://pathcrawler-online.com>)
 - パス解析から, テストケースの自動生成
 - 04年から公開している.
- *DART(Directed Automated Random Testing)*
 - ソルバーで解けない制御論理をランダムテストで解く
 - Godefroid, Patrice; Nils Klarlund, Koushik Sen (2005). "DART: Directed Automated Random Testing". Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. New York, NY: ACM. pp. 213–223. ISSN 0362-1340. Retrieved 2009-11-09.
 - http://research.microsoft.com/en-us/um/people/pg/public_psfiles/pldi2005.pdf
- *EXE(KLEE)*
 - Dawson, Engler; Cristian Cadar, Vijay Ganesh, Peter Pawloski, David L. Dill and Dawson Engler (2006). "EXE: Automatically Generating Inputs of Death". Proceedings of the 13th International Conference on Computer and Communications Security (CCS 2006). Alexandria, VA, USA: ACM.
- 他にも多くの研究

- *Pathcrawler-online.com*
 - オンライン上で処理を公開している
 - Tool自体は非公開
- *CUTE and jCUTE*
 - 研究用に限り, バイナリーで適用される
- *CREST (CUTEの拡張)*
 - オープンソースとして公開
- *CATG, Jalangi (Java向け)*
 - オープンソースとして公開
- *Microsoft Pex*
 - Visual Studio 2010に含まれる
- *Wikipedia の concolic testing 参照*

- 条件分析しなくてもテストケースが自動生成され、実行される
- テストの結果はプログラマが意図したとおりに動作するかどうか
- 問題は仕様との一貫性
- 評価のためには仕様から入力と出力の関係を作成し、照合する必要がある

静的解析は仕様をテスト出来ない

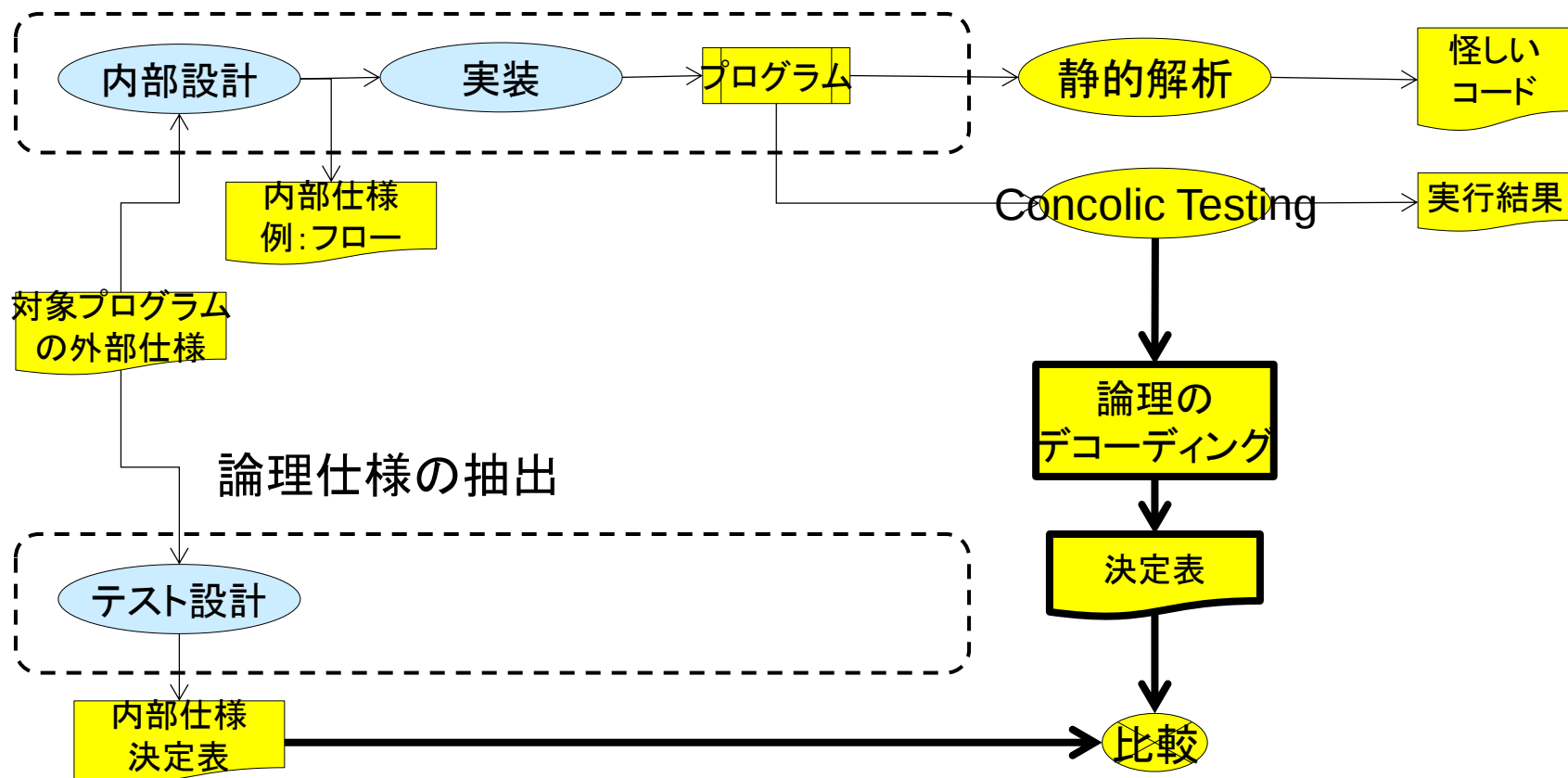
- 静的テストは、設計多重による検算では無い
- 検出は
 - 蓄積したナレッジと比較
 - Concolic 実行による異常な動作



決定表による論理検証(CFD++の狙い)

- Keiji Uetsuki, Tohru Matsuodani, Kazuhiko Tsuda (2013). An Efficient Software Testing Method by Decision Table Verification, International Journal of Computer Applications in Technology Vol. 46, Issue 1, 54-64

設計活動



- 動的テストを実行しない:
 - テストデータの作成不要, テスト環境不要
- 現場におけるテスト課題を改善する
 1. 漏れが無いこと……2.3 漏れ2は解決, 漏れ1はプログラムとの多重化
 2. 無駄のない少ない量…2.6 過剰なテストは, 存在しない組合せ削減
 3. 属人性……………自動化率が高いので改善
- テスト実行における手間(工数, コスト)……ほぼゼロ



- テストコスト削減は依然、ソフト開発の大きなテーマ
- CFDはテスト方法合理化による適正なコスト削減手段
 - *それを実現しやすくするのが論理構造設計*
 - *論理構造解析からテストケース迄、決定表で実施する方法の提示*
- テストケース設計自動化の進展への対応
 - *Concollic Testingの登場と実用化*
 - *決定表を使った仕様情報と実装情報を比較可能にする手段開発*
- CFD++の今後
 - *CFD++を利用したテストプロセス像の表現と提示*

この資料は(有)デバッグ工学研究所によって、今回の教材として個別に開発されたものです。この資料の複製、無断使用を禁止します。2013年9月

デバッグ工学研究所は、テスト、デバッグの様々な問題解決を支援します。

HP: <http://www.debugeng.com/> 問合せ: CFD@debugeng.com