Concolic testingと背景技術
~テスト技法の新動向~

㈲デバッグ工学研究所
松尾谷　徹
matsuodani@debugeng.com

R-13. 9

概要と目次

■ 概　要

☆ 現場におけるテストの課題：テスト漏れ，テストケース数の爆発，属人性（今も昔もかわらない）

☆ テストの方法：　動的テストが主流でその技法の進歩は緩慢

★ 新たな技法：静的解析（静的テスト）の台頭

★ 活発な基礎研究（海外）とオープンソース（ツール）

★ 「現場におけるテストの課題」を解決する薄明り

■ 目　次

1. テストの領域

- ソフトウェアテストの歴史

ソフトウェアテスト

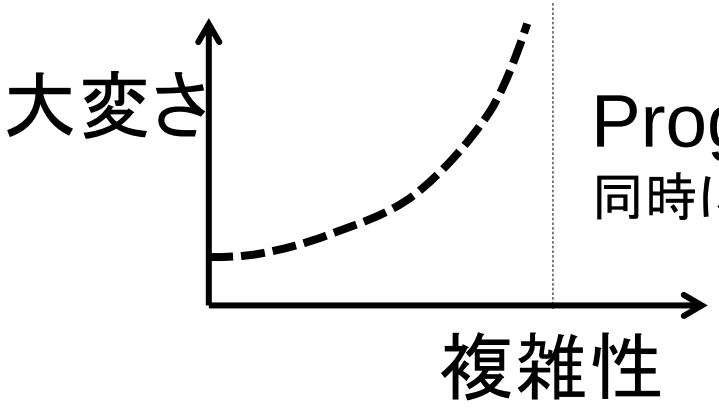
- *Software Engineering (SE)　の始まり*
- *NATO防空システム開発の失敗　－＞　その対策*
  - ソフトウェア開発・失敗の共通的な現象
    - テスト段階で，さまざまな不具合，機能不足，・・・テストで混乱
  - その原因は
  - 色々あるが　＜組織化，工業化されない，経験学習型プログラマーの群＞
- *そこから，2系統*
  - ソフトウェアが持つ構造や変換など：アーキテクチャ，コンパイラ，・・・
  - ソフトウェア関連の活動や管理など：プロセス，品質特性，・・・
  - 要するに，日曜大工的な経験手法から，工学的な根拠に基づく手法へ
- *その中で，テストは？*
  - 取り残された領域

テスト（デバッグ）の出現

- *プログラミングと同時*
- *世界初のコンピュータ　ENIAC　（1946年）*
- *その試運転用のデモ用プログラムが動作しない！・・・・世界初のデバッグ*
  - 当時のプログラムは，コード（電線）で結線する方式

Programming と Debugging　＜表と裏の一体＞

- *人が，複雑な論理を取り扱うこと＝＞不慣れで意図した通りにならない*
- *複雑性（量的，質的）が増すと，デバッグが大変*



Programmingの限界（表の世界）
同時にDebuggingの限界（表裏一体）

1.2 Programmingの進化

限界打破のために

- *複雑性を緩和する，工学的なアプローチ： ソフトウェア工学*

例

- *ENIAC 初期のProgramming パッチパネルの配線*
  - Programmingは大変 デバッグは1ステップ実行で確認
  - フォン・ノイマンの提案方式に変更 Programming は数日から数時間へ改善
- *「機種固有の奇怪な機械語」から，プログラミング言語（標準化）へ*
  - 1950年代から出現 （57年 Fortran 60年 Cobol 62年 Lisp ）

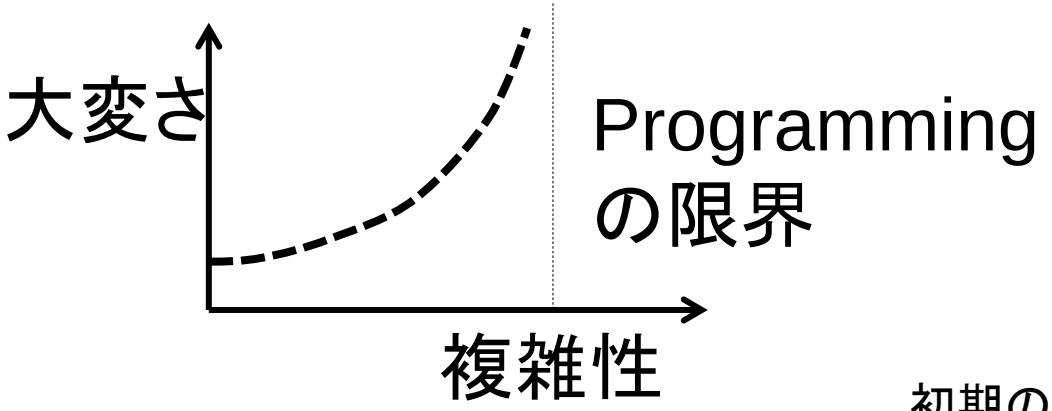
限界の原因要素と対策

- *質的： ①実装機能の論理 ②目的機能の論理*
- *量的： ①実装機能の処理 ②目的機能の処理*

  - 実装機能： コンピュータを使うための機能（当初は機械語へ変換）
  - 目的機能： 意図した機能（要求仕様）
  - 論理： 条件・判定 処理： 入力を出力に変換

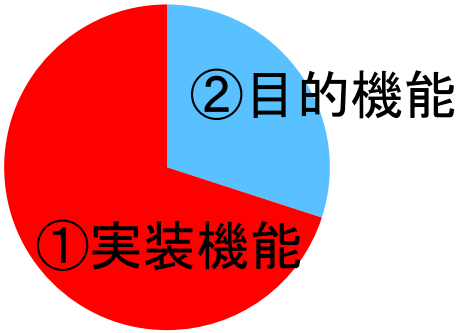
1.3 ソフトウェア工学(例:コンパイラー)の効果

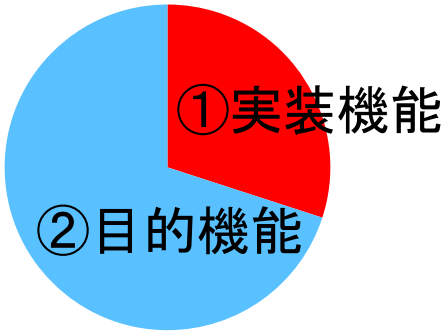
- 実装機能の割合は減少　しかし限界自身は残る



構成
が
変化

初期のソフトウェア工学は,
コンピュータを使うために生じる
実装機能の複雑性を緩和した.





使いやすくなると,
目的機能の規模拡大が
続き, 再び限界に達した.

DEI

設計の原理：分解（analysis）による設計

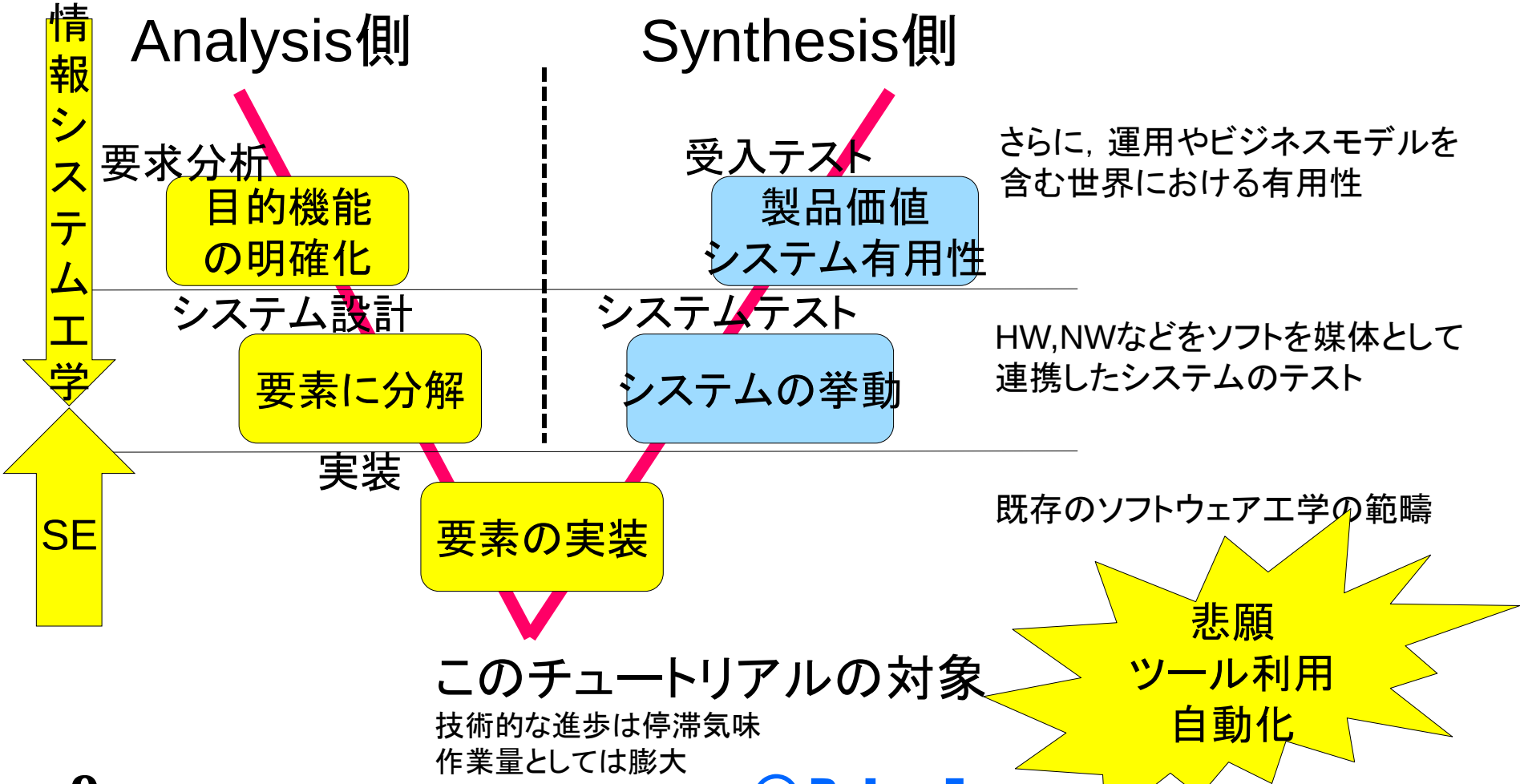
- *人が取り扱える大きさに分割 （要素とインタフェース）*
  - 人がProgramming / Debugging 出来る限界内の粒度に分解する.
- *分解の進め方として，幾つかの方法論が出現*
  - 機能（処理）指向設計， データ指向設計，オブジェクト指向設計
  - 要素の再利用
- *ユニット，関数，メソッド・・・呼び名は様々だが，規模を小さくすると 実装における Programming / Debugging は楽になった.*

手作業そのものは変わらず

- *コーディングそのものは，30年以上変わらず. （流用の可能性は増大）*
- *単体テスト，統合（結合）テストそのものは，30年以上変わらず.*

DEI

- 統合すると現われる不具合の増加



1.6 テストの主流 Dynamic Testing

- *一般的なテストであり，次の手順で行われている*
  1. テストケースを設計する
  2. テストケースから具体的なテスト入力を作成 ＜テスト入力＞
  3. テスト結果が満たす条件を求める ＜想定結果＞
  4. テストを実行し，＜実際のテスト結果＞と＜想定結果＞を比較する
- *テストケースの設計によってテストの網羅性が決まる*
- *テストケースの設計方法*
  1. 仕様ベース： テスト対象に対する外部仕様を基に設計する
  2. 構造ベース： 対象の内部仕様から制御構造 and/or データ構造を基に設計
  3. 経験ベース： 過去の事例からバグの推測を行い設計する
- *属人的な活動*
  - レビューと同様，手順は明らかだが，出来栄えは作業者のスキルでバラツキ
  - 再現性が低い活動 ＝＞ ツール化，自動化が不足（難しい）

2. 動的テストとその課題

- 主流となっている動的テストの原理
- その課題

動的テスト(Dynamic Testing)

- 動的テストは，経験から始まった技術

分類：多様な現場において発展したため分類方法すらも多様

- *仕様ベース/構造ベース/経験ベース　例　ISO/IEC 29119-4*
- *機能ベース(Functional)/非機能ベース(Non-Functional)*
- *レベルベース：単体テスト，統合テスト，システムテスト*

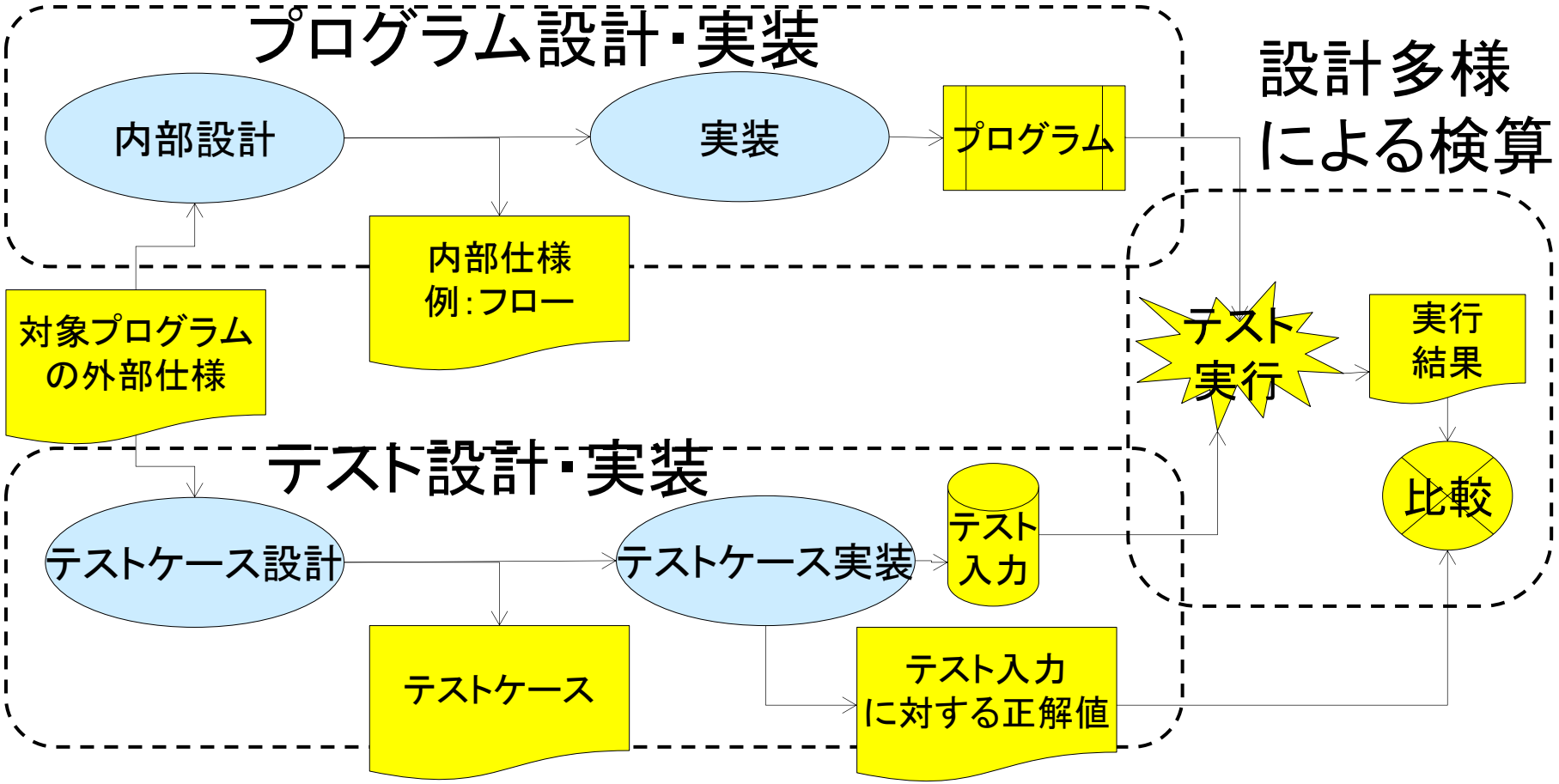
課題：

- *経験的で再現性が低い*
- *工学的なアプローチが不足し，自動化が困難*
- *その結果産業界：*
  - 質的リソース不足：　テストケース設計
  - 量的リソース不足：　テスト実装とテスト実行
  - テストケース設計とテスト実装の分離が困難

- *課題（テスト漏れ，テストケース数の爆発，属人性）について整理*

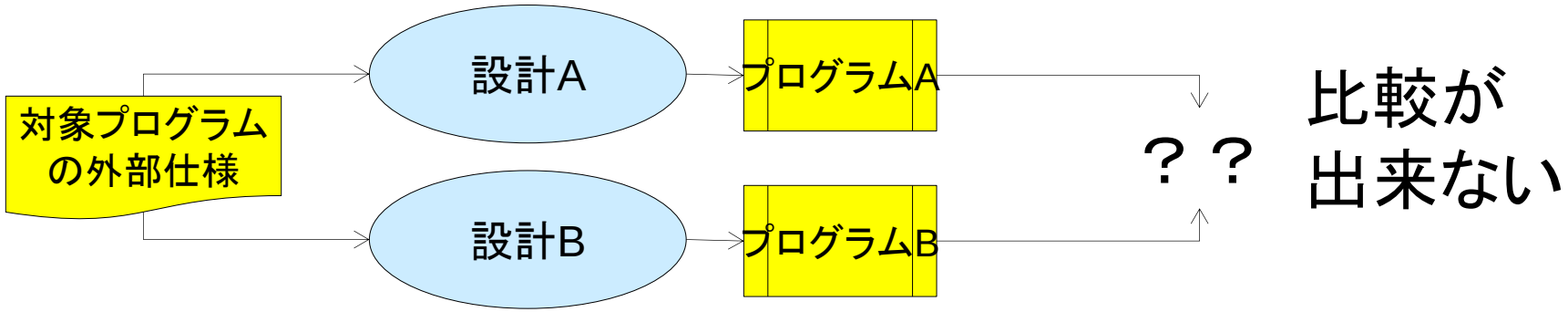
2.1 動的テストの原理：設計多様による検算

- 設計多様： 異なった設計方法による比較

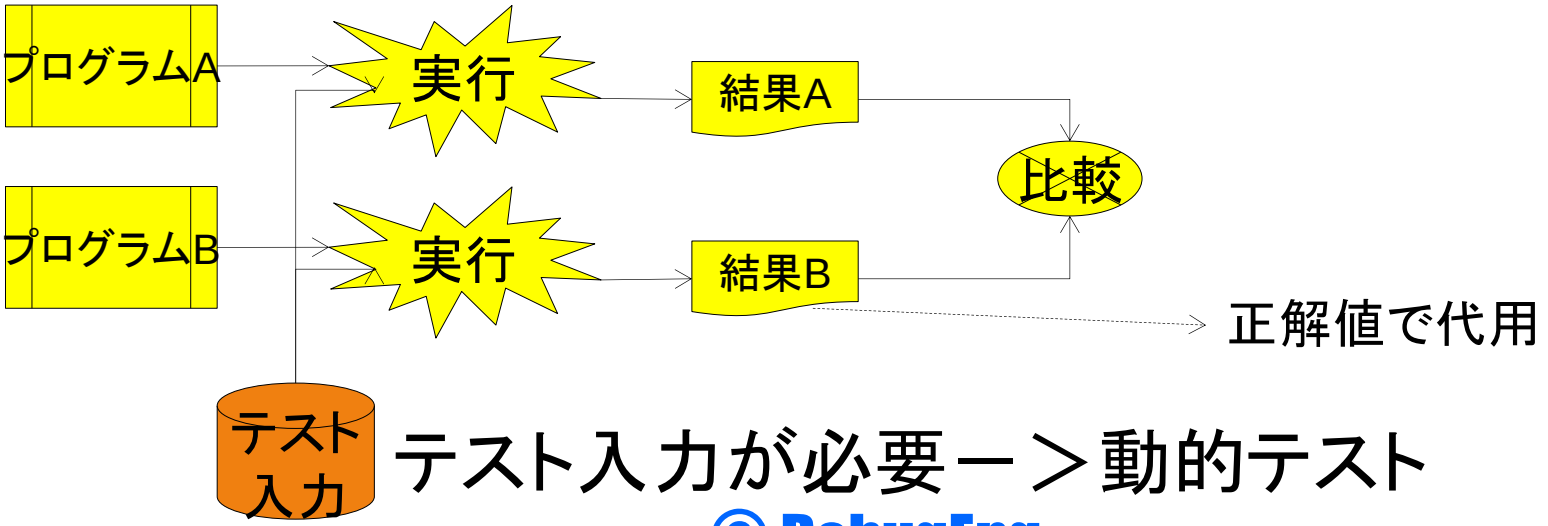


2.2 設計多様が必要な理由

- *別の方法：設計多重による検算*

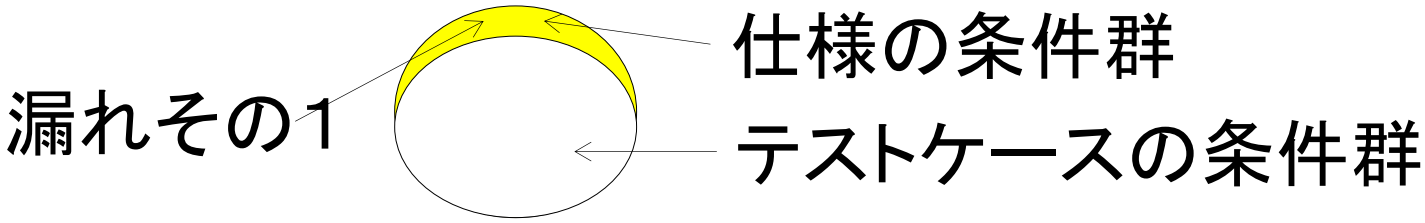


- *設計の結果（プログラムA,B，・・）を比較しても検算にならない*
  - 設計A,Bが正しい場合でも，プログラムは同じにならない
- *そこで動作させて比較する：　テスト入力を作る必要がある*

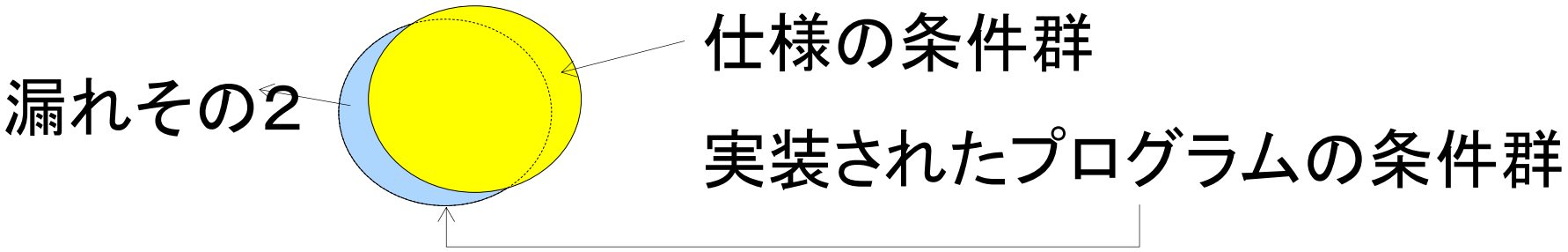


2.3 課題1:テストケースの漏れ

- 前提: テスト対象の外部仕様は正しい(仕様の漏れ, 誤りは対象外)
- テストケースの要素: 条件と処理
  - テスト入力: 条件を満たす入力域
  - テスト結果: 処理の結果
- テストケース漏れ:仕様の条件群中, テストケースの条件群に無いもの



- 冗長なプログラムの条件群



2.4 漏れその1(テストケース抽出の漏れ)

- 仕様から，漏れなく条件群を抽出する
- 条件群は2種類

1. 独立の条件： テスト技法としては，同値分割
2. 他の条件と関係を持つ条件： テスト技法としては原因結果グラフなど

- 漏れが生じる主な原因は，

1. 仕様の条件に関する記述が曖昧なことから生じる漏れ
   - 設計技法の限界＋設計時の施工問題
2. 抽出の技法不足や網羅の確認が不備で生ずる漏れ
   - テスト技法の限界＋その施工問題

＜メモ＞

- 形式仕様技術は，伝達ミスを防止する
- そこまで厳密でなくとも，決定表など条件群を記述する技法はある
- 「自然言語で制約なしで記述する」エンジニアとしては有り得ないが現実

2.5 漏れその2(仕様に記述されていない機能)

- 仕様から観ると，仕様に無い冗長な条件や処理： バグとは限らない
- 実装時仕様と呼ばれる部分： 例 メモリー解放，I/O後処理
  - 実装機能に関連する条件や処理
  - 目的機能が失敗した場合の後処理など
- 仕様には記述されていないので仕様ベースのテストでは検出できない
- 構造ベースのテストと併用すると，検出できる

＜メモ＞

- 使用する言語と動作環境(os等)によって差があるが
  ソースコードの20～30％を占める
- 実装機能(コンピュータを使うための機能)に関連する
  - 目的機能からは導かれない

2.6 課題2：過剰なテストケース

- 条件群には，独立で無い条件群が存在する
- 例　割引：　①60歳以上は20%割引
  ②毎月20日は，女性の場合に限り25%割引
  ①②の組合せ：　②を優先する
  - テストは4ケースが必要
- 無則（仕様では定義されない条件）の条件の組合せで生ずるバグ
- 例　③クーポン割引，④会員割引，⑤お誕生割引，⑥住民割引，・・・⑳・・
  割引に関係しそうなもの20個
  テストケースの数は2の20乗　（1,048,576:100万の桁）
  テストケース数の爆発

- ブラックボックステストの場合：　原理的にすぐ爆発する
- ホワイトボックステストの場合：　直列の条件文は，組合せが積になる
  - そこで，C0,C1程度以上のテストを行わない（網羅性を抑える）

2.7 課題3:属人性

- テスト設計の出力ー＞テストケース
- それが「正しいか，漏れていないか，過剰か」を客観的に計測できない
- テスト設計の評価が出来ないので，属人性が残る
- 教育・訓練不足も，同様な原因（知識とテスト成果の相関が低い）

- 代用特性
- 規模との相関（過去の統計量）
- バグ成長曲線（信頼度成長曲線）
- パス網羅率
- バグ埋め込み

2.8 テストの課題(動的テストの課題)解決の方向

DEI

- 個々の課題に対して様々なアプローチがある
  1. 要求仕様自身の妥当性を実装前に評価:**モデルベース開発**
  2. 設計仕様自身の曖昧さを解決し，コードの生成:**形式仕様と自動生成**
  3. 設計仕様のアルゴリズムの正しさ検証:**モデル検証**
  4. 仕様とは別に，実装上の問題を見つける:**静的テスト**
  5. 設計仕様と共に実装したコードの振舞いを定義:**テスト駆動開発**
- **間接的なアプローチ(プロセスや管理技術)は多々あるがここでは除く**

- **対策は色々あるが，**
- **導入障壁(コスト，スキル，過去の資産，・・)のため現場のテストが合理化される傾向は見えない**

3. 静的解析（静的テスト）の進歩

- 近年，実用化が始まった

3.1 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*

3.3 CIL（C Intermediate Language)の出現

DEI

静的解析のための前処理

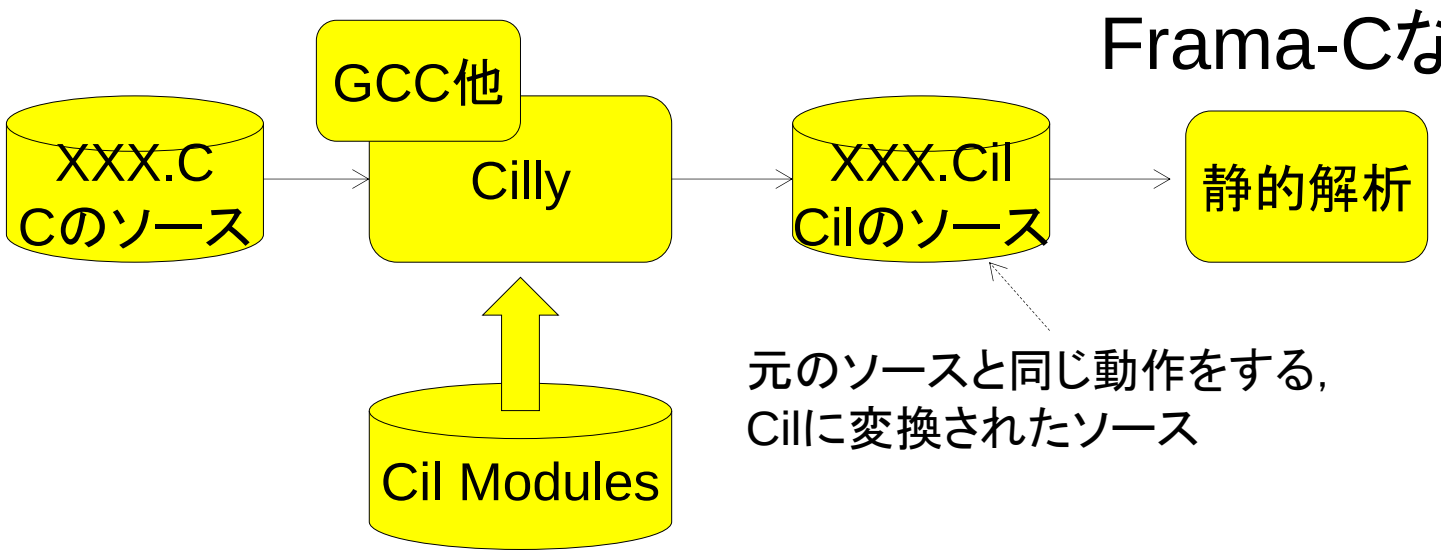
- *C言語を始め，プログラミング言語は曖昧な表現を含んでいる.*
  - 機種互換や最適化において問題
- *そこで，厳密な表現に変換する技術*
- *CILは，制御構造を解析し，抽象構文木(abstract syntax tree ）で表現することにより曖昧性を排除する.*
  - George C. Necula, etc ；CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs，CC ‘02 Proceedings of the 11th International Conference 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で公開されている*
  - 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）

C->CIL 変換 ->静的解析

- *概要*



Control-Flow Graphs
Data flow analysis framework
Inliner
Points-to Analysis
StackGuard
Heapify (ローカルアレイ処理)
One Return
etc

Ocaml で追記
様々なチェック

C－＞Cil
変換のルールを記述
詳細は
http://kerneis.github.io/cil/doc/html/cil/

CILの制約

- 入力となるCのプログラムは，コンパイルエラーを含んではいけない（当然）
- コメントが引き継がれない
  - プラグマ（コンパイラーに情報を渡すC言語のメソッド）が使える
- 全てのローカルスコープを関数スコープに変換するので，初期化を伴う宣言は，分離される．そのため const qualifier　は使えない．

- 他にも，Bugによる制約は，**Known Bugs and Limitations**
  - http://kerneis.github.io/cil/doc/html/cil/

3.5 Frama-C

- *formal verification（形式的検証）と静的解析のためのフレームワーク*
- *オープンソース：* http://frama-c.com/
- *特徴：静的解析のための各種プラグインが利用可能*
  - 3種類の基本となるプラグイン
  - Value analysis plug-in： プログラム中の変数のvariation domainsを求める
  - Jessie plug-in： ACLS(ANSI/ISO C Specification Langage)による検証
  - WP plug-in： WP（weakest precondition 弱い事前条件）の検証
  - これらを応用したプラグインが提供されている
  - 例：
    - Impact analysis： 変更の影響分析
    - Scope & Data-flow browsing：データフロー解析
    - Slicing： スライシング
    - Spare code： 使われないコード検出
    - PathCrawler ： テストケース生成

3.6 Coverity社の成功

大学発のベンチャー企業

- *2003年　スタンフォード大にて設立*
- Coverity SAVE ：Static Analysis Verification Engine
  - 静的解析検証エンジンを中心に事業を拡大した
  - 検出が難しいバグやセキュリティホールを指摘
  - 誤検出が少ない
- *多種言語サポート：　C, C++, C#, Java*
- *優れたScalability：　大規模なプログラムにも対応できる*
- *無駄な指摘が少ない*

- *動的な振舞いの解析を行う Coverity Dynamic Analyzerも開発*

4. Concolic Testingの誕生

4.1 Unit testing の技術として出現

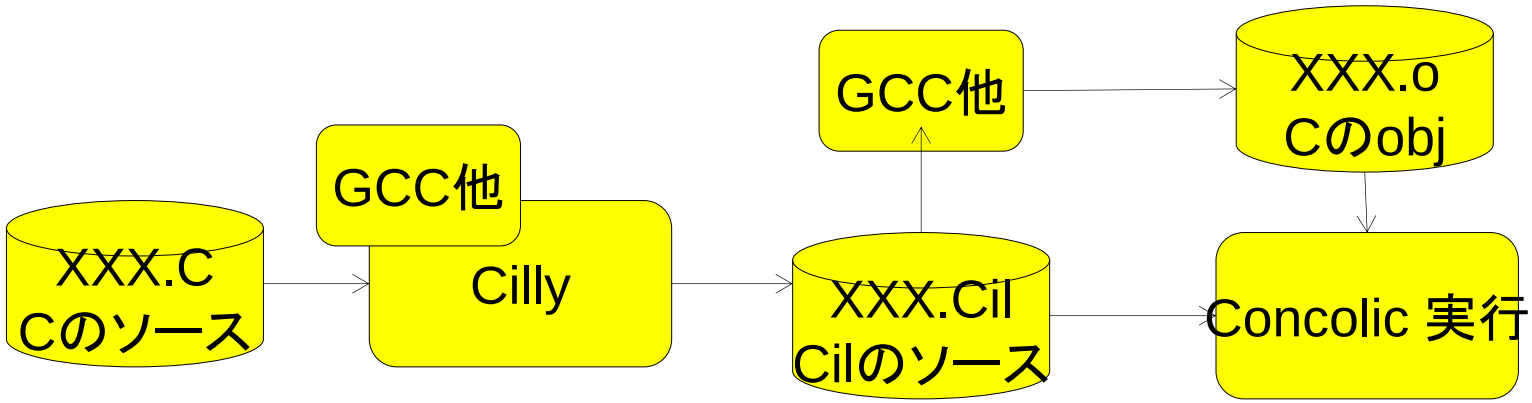
- *CUTE: concolic unit testing engine*
- *この中で, Concolic testing なる用語が使われた*
  - Sen, Koushik; Darko Marinov, Gul Agha (2005). "CUTE: a concolic unit testing engine for C". Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations 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

4.2 シンボリック実行

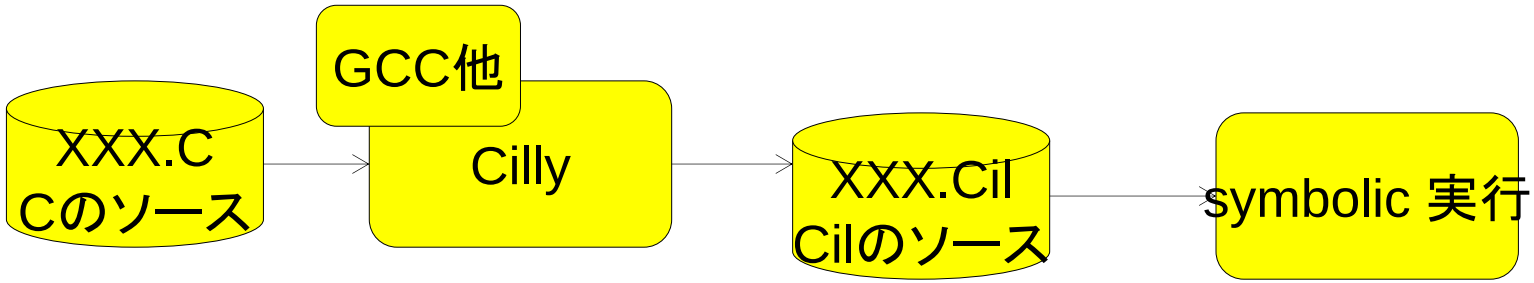
- *ソルバーで解くために*
  - 網羅すべき制御が実行されることを確認することを確認する
  - そのために，シンボリック実行が必要
- *symbolic execution の改善*
  - C言語の動作を完全にインタプリタするのはとても困難
  - そこで，Cillyによるフロント処理済み（AST化）の中間言語（Cil)を対象とする
  - さらに，一部の実行はpiggy-back方式で行う
  - concrete（execution）and symbolic execution ＝Concolic
    - Concrete execution :実際の実行（symbolic execution では無い普通の実行）
  - 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

処理の流れ

- *Concolic Execution*



- *Symbolic Execution*



4.3 他のアプローチ

- *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*

4.4 利用可能な tools

- *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*

6. Concolic Testingの応用例

- テストの課題を解決するには

6.1 現場におけるテストの課題 再確認

- 静的解析の進歩
  - *しかし，仕様ベースのテストではない： 仕様通りに作られているのか？ ×*

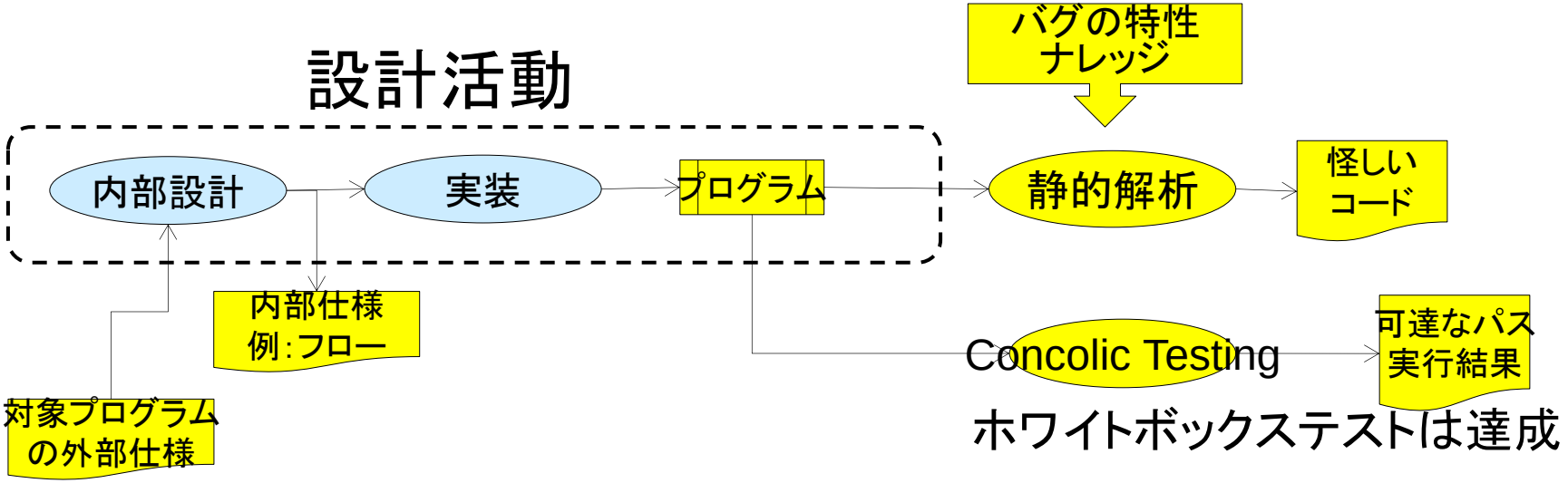
- 現場で必要なことは，

1. テストの品質
   1. *漏れが無いこと*
   2. *無駄のない少ない*
   3. *属人性*
2. テストの納期，コスト改善

- 静的解析の成果を動的テストへ！！

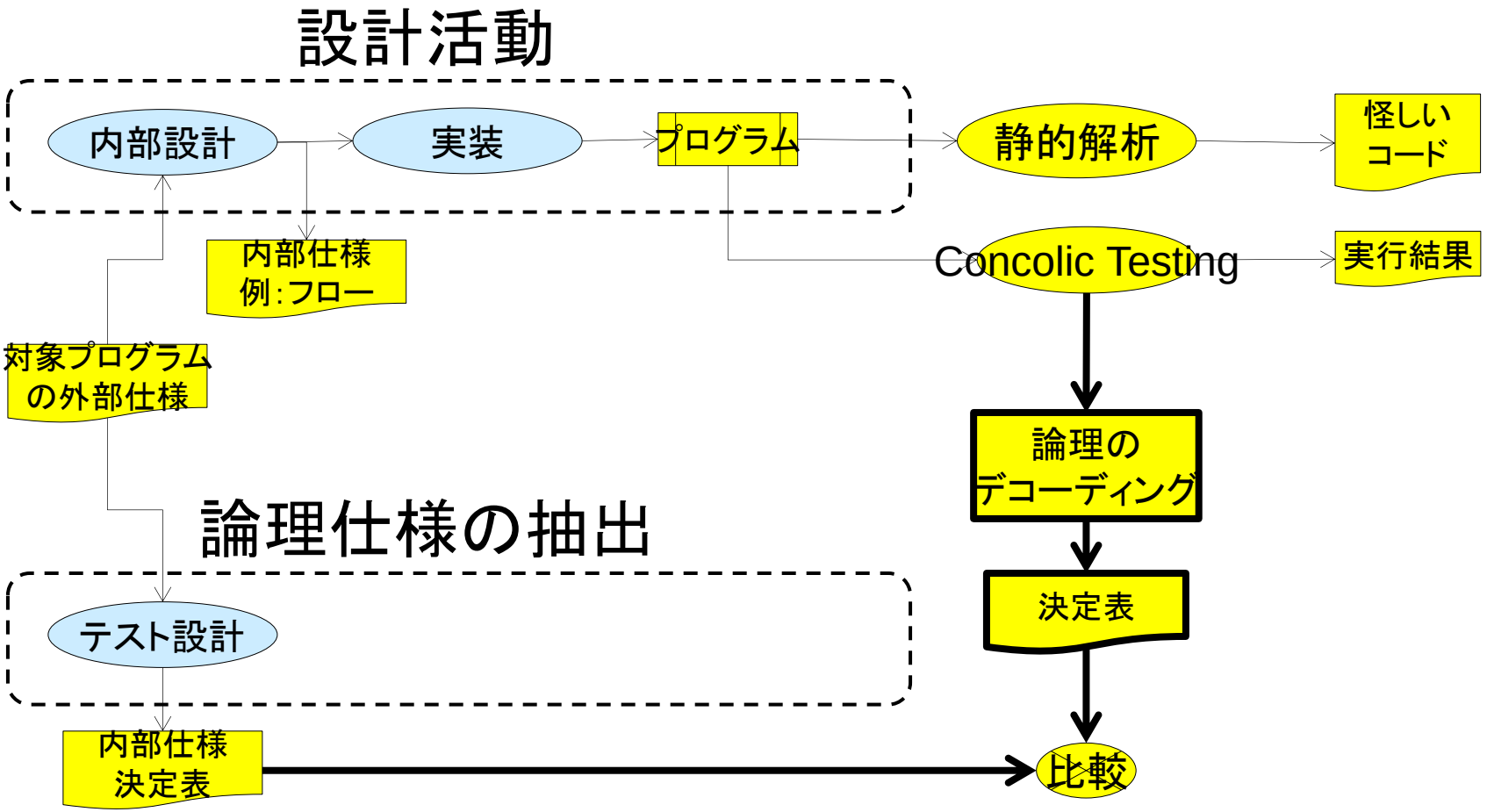
6.2 静的解析は仕様をテスト出来ない

- 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



6.3 決定表による論理検証

– 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

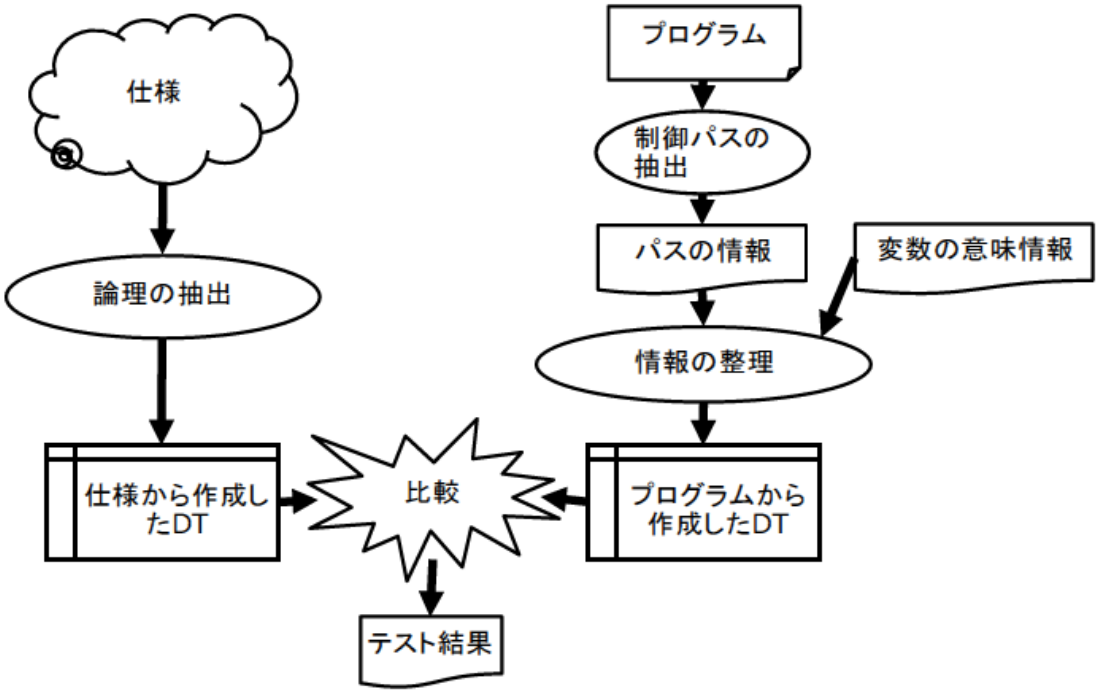


特徴

- 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

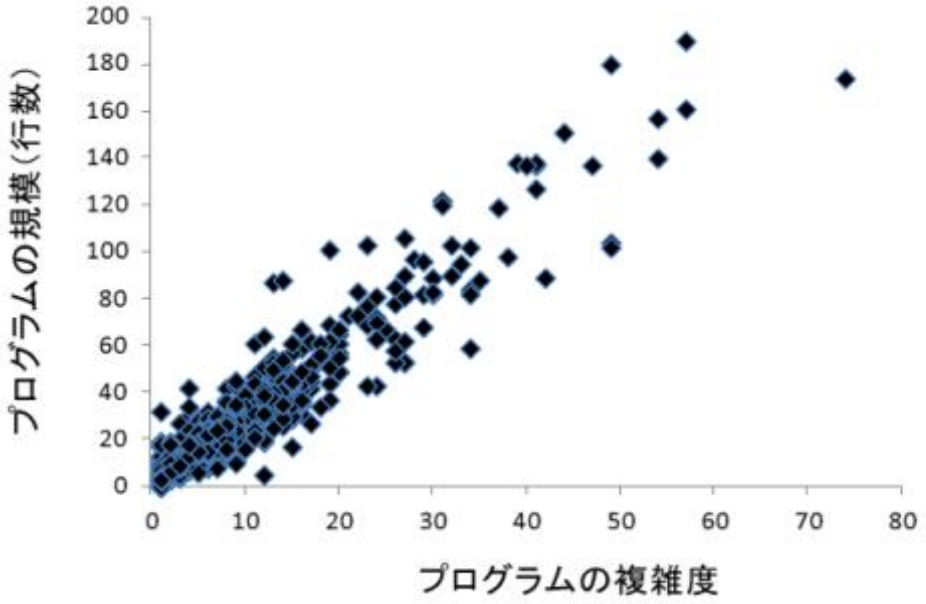
1. 漏れが無いこと・・・・2.3　漏れ2は解決，漏れ1はプログラムとの多重化
2. 無駄のない少ない量・・2.6　過剰なテストは，存在しない組合せ削減
3. 属人性・・・・・・・・・・・・・自動化率が高いので改善

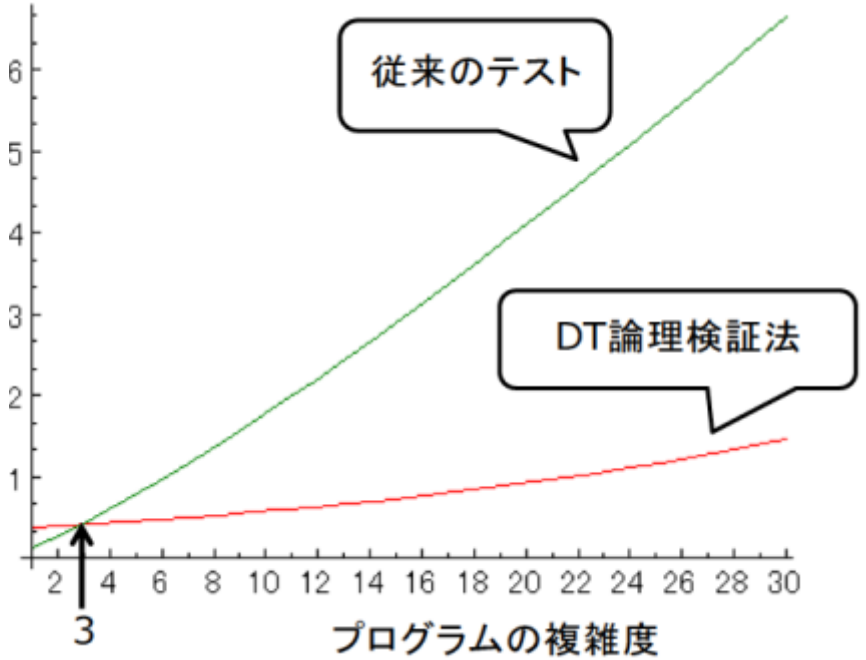
- テスト実行における手間（工数，コスト）・・・ほぼゼロ



6.4 決定表による論理検証の効率の比較

- 簡単なプログラム（複雑度が低い）は，従来のテストで十分
  - 簡単なのでテストケース数が少なくテスト漏れも少ない
- 複雑度3以上になると，DT論理検証の方がコストが低い
- モジュールの数，複雑度の分布は，対象によって異なるが，事例では，単体テスト全体のコストを半減





6.5 目的機能の論理と実装機能の論理

- プログラムの論理
  - *（目的機能＋実装機能）を実現する論理*
  - *作られたプログラムから，両者を分離するのは困難*
  - *静的解析による解析は可能*
- プログラム設計
  - *目的機能が与えられる*
    - 論理：ビジネスルール，制御ルールとして決定表で表現可能
    - 処理：入力を出力に変換する仕様として定義可能
  - *実装機能は，設計段階で考慮 （データー構造，制御構造，メモリ管理，他）*
- 決定表による論理検証を使ってみると
  - *仕様から決定表を作成する問題*
    - 目的機能の論理は，仕様から抽出できる
    - 実装機能の論理を仕様から抽出するのは難しい

- 改造時のインパクト解析
  - *プログラムを修正した時，プログラムの振舞いを解析する*
  - *改造前の静的解析と改造後の静的解析*
    - 可達なパスとそのテスト入力，結果は生成できる
    - 何をもって，インパクトとするのか？　悪影響とは何か？
- テスト駆動開発への応用
  - *テスト駆動開発の目的：　開発者が設計物を早く正確に理解する効果*
  - *網羅的なテスト設計が不足*
    - 品質保証の目的で行うテストは，別に行っている
    - 静的解析の応用による合理化

＜有望＞

- 新規性のある応用研究テーマ
- 実務の合理化ツール

8. まとめ

まとめ

- ソフトウェア工学におけるテストの分野・・・・・未開拓分野
- 動的テスト技術進歩は緩慢・・・・・・30年ほど変わらず
- しかも，現場の世界において大きな作業量がある

- 静的解析は，ここ10年で大きな進歩
- しかし，静的テストは，現場のテスト作業改善にさほど役立っていない

- そこで，現場のテストを改善する応用研究の意味がある
- 事例として，決定表による論理検証を紹介した
- 情報システム・信頼性研究会（信頼性学会）ではこの課題をとりあげている
  - 特に，保守，改造におけるデグレードテストの効率化
- 研究者： 新規性のある応用研究分野
- 実務者： ツール活用による実務の効率化
- 課題： 目的機能の論理，実装機能の論理を表現する方法
  - 一つの案 ：古くからある決定表の活用

ご清聴ありがとうございました

松尾谷 徹
matsuodani@debugeng.com