JaSST'15 Tokai

はじめてのコンコリックテスト

～基本原理から知るホワイトボックステストの新技術とその応用～

2015年11月6日　植月啓次

はじめに

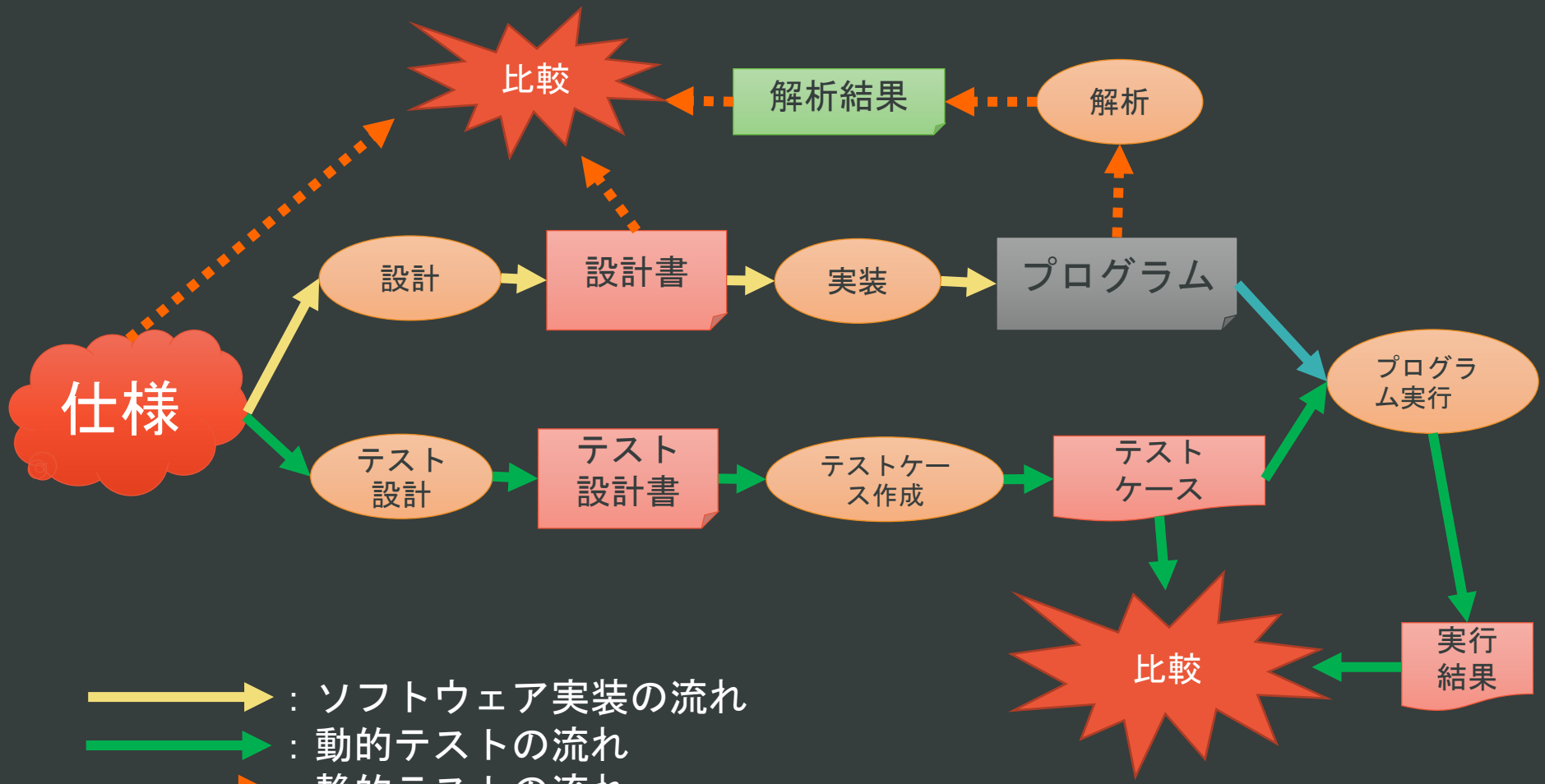
- コンピュータの処理能力の飛躍的な向上により、プログラムコードにたいする解析技術は大きく進化しています。
- 近年、実用段階に入り盛り上がりを見せている解析技術およびその応用技術が、コンコリックテスト(Concolic Test)です。
- 本セッションでは、コンコリックテストの基本原理、事例の紹介、ツールのデモ、応用のアイデアについて解説します。

もくじ

- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

もくじ

動的テストと静的テスト



もくじ

- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

シンボリック実行（Symbolic Execution）とは

- プログラムを解析し、存在する実行パス（実行可能なパス）の抽出およびそのパスを実行するための入力データを決定する技術
- 入力データを「シンボル」として扱う
- 「実行」とあるが、実際にプログラムを実行するわけではない

シンボリック実行の中身

- 例　：２つのパスをもつプログラムにたいする解析

```
function(入力 x)
{
    // yはローカル変数

    y = 2 * x

    if (y == 10)
        print(“FAIL”)

    print(“PASS”)
}
```

シンボリック実行の中身

■例　：２つのパスをもつプログラムにたいする解析

```
function(入力 x)
{
    // yはローカル変数

    y = 2 * x

    if (y == 10)
        print(“FAIL”)

    print(“PASS”)
}
```

シンボル s
で入力をおきかえ

```
function(入力 s)
{
    // yはローカル変数

    y = 2 * s

    if (2*s == 10)
        print(“FAIL”)

    print(“PASS”)
}
```

シンボリック実行の中身

■例　：２つのパスをもつプログラムにたいする解析

２つのパス：
　2*s = 10 が成り立つ（True）
　2*s = 10 が成り立たない（False）

s が 5 のときと、5 以外のとき

2*s = 10 が成立する s があるかどうかを判定する問題
　　「充足可能性問題」

※ 2*s = 10 を「制約」と呼ぶ

```
function(入力 s)
{
    // yはローカル変数

    y = 2 * s

    if (2*s == 10)
        print(“FAIL”)

    print(“PASS”)
}
```

充足可能性問題（Satisfiability problem, SAT）とは？

- 命題論理式を対象にして、式中の変数のTrue/Falseの組み合わせで全体をTrueにすることができるかどうかを判定する問題

変数Ａ，Ｂ，Ｃを含む論理式

（Ａ∧¬（Ｂ∨Ｃ））∨（¬Ａ∧¬Ｂ）

これをTrueにするための変数のTrue/Falseの組み合わせはあるか？

∧：論理積（ＡＮＤ）
∨：論理和（ＯＲ）
¬：否定（ＮＯＴ）

充足可能性問題（Satisfiability problem, SAT）とは？

■命題論理式を対象にして、式中の変数のTrue/Falseの組み合わせで全体をTrueにすることができるかどうかを判定する問題

変数Ａ，Ｂ，Ｃを含む論理式

（Ａ∧¬（Ｂ∨Ｃ））∨（¬Ａ∧¬Ｂ）

総当たりの真理値表を書いてみると、わかる

A	B	C	全体
T	T	T	F
F	T	T	F
T	F	T	F
F	F	T	T
T	T	F	F
F	T	F	F
T	F	F	T
F	F	F	T

充足可能性問題（Satisfiability problem, SAT）とは？

■命題論理式を対象にして、式中の変数のTrue/Falseの組み合わせで全体をTrueにすることができるかどうかを判定する問題

変数Ａ，Ｂ，Ｃを含む論理式

（Ａ∧¬（Ｂ∨Ｃ））∨（¬Ａ∧¬Ｂ）

A	B	C	全体
T	T	T	F
F	T	T	F
T	F	T	F
F	F	T	T
T	T	F	F
F	T	F	F
T	F	F	T

総当たりの真理値表を書いてみると、わかる

コンピュータで答えを導く手法（アルゴリズム）がさまざまあり
一般にそのアルゴリズムを実装したソフトウェアを「SATソルバ」という

シンボリック実行によるパスの抽出と入力データの特定

```
function(入力 a, b, c)
{
    // x はローカル変数

    if (a) {
      x = 1
    }
    if (b > 9) {
      if (!a && c) {
          x = x+1
      }
      x = x+2
    }

    Return x
}
```

シンボリック実行によるパスの抽出と入力データの特定

```
function(入力 a, b, c)
{
    // x はローカル変数

    if (a) {
      x = 1
    }
    if (b > 9) {
      if (!a && c) {
          x = x+1
      }
      x = x+2
    }

    Return x
}
```

シンボル α、β、γ
で入力をおきかえ

```
function(入力 α, β, γ)
{
    // x はローカル変数

    if (α) {
      x = 1
    }
    if (β > 9) {
      if (! α && γ) {
          x = x+1
      }
      x = x+2
    }

    Return x
}
```

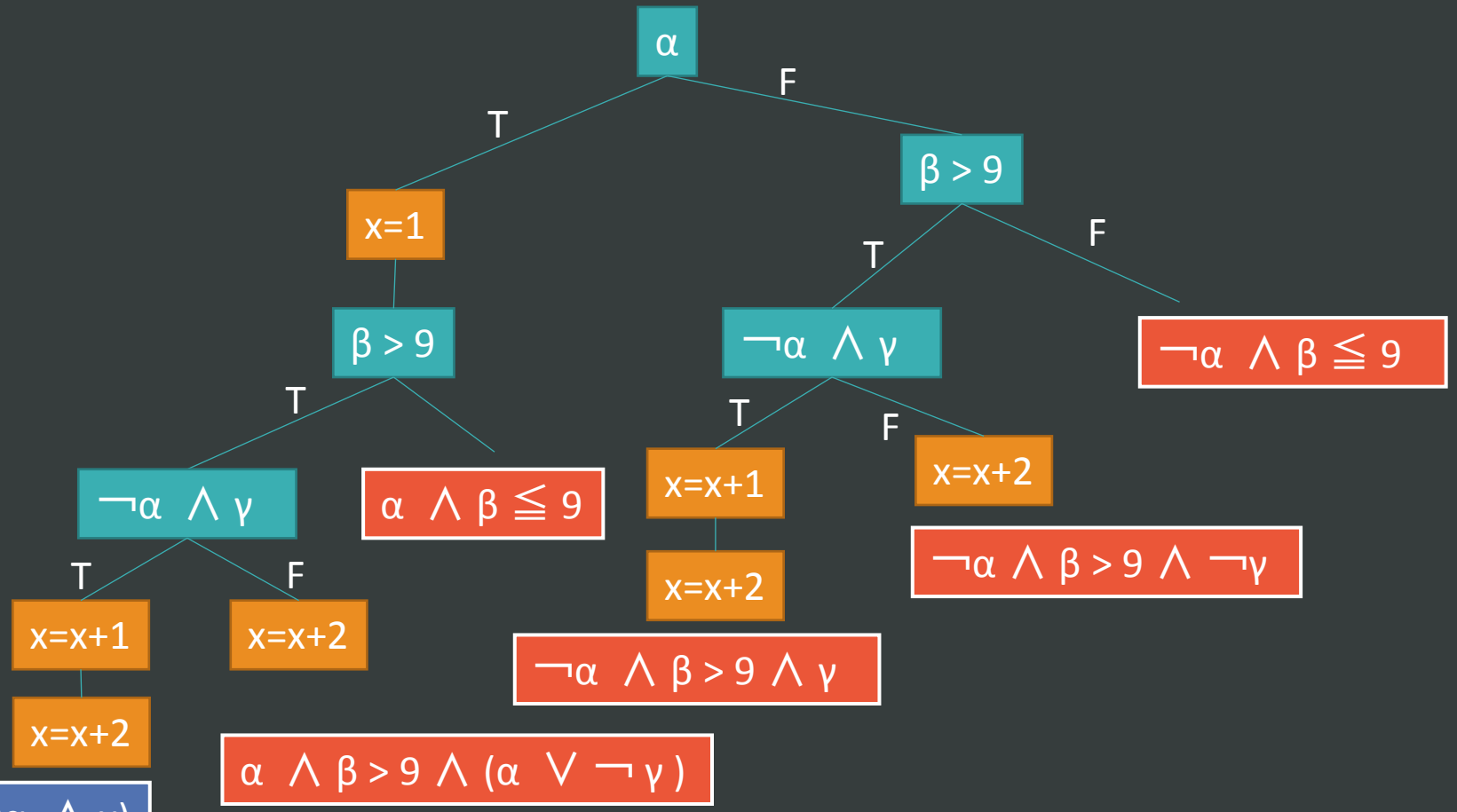
シンボリック実行によるパスの抽出と入力データの特定

```
function(入力 α, β, γ)
{
      // x はローカル変数

      if (α) {
        x = 1
      }
      if (β > 9) {
        if (! α && γ) {
            x = x+1
        }
        x = x+2
      }

      Return x
}
```

Returnに至るまでのパスを解析する。



シンボリック実行によるパスの抽出と入力データの特定

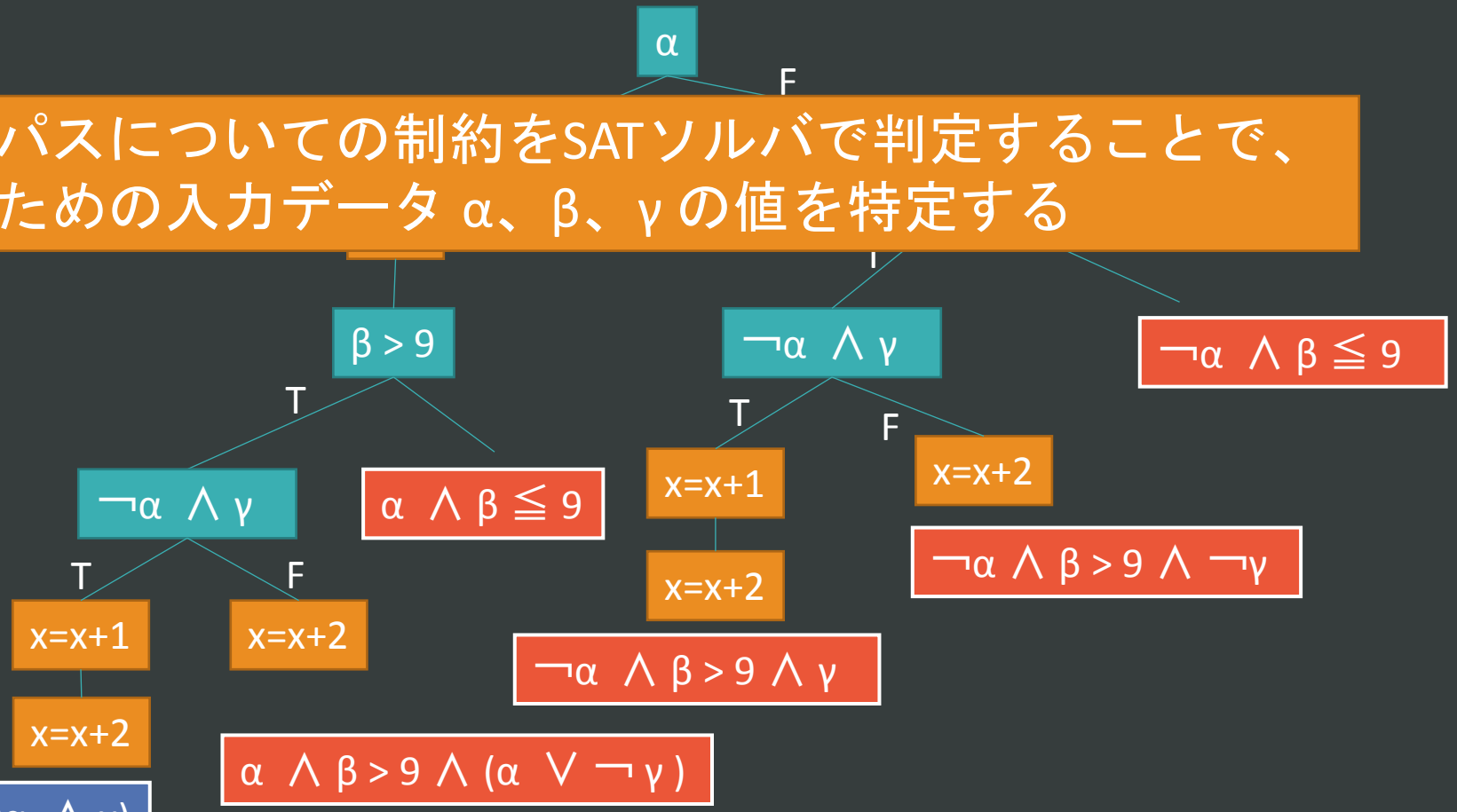
```
function(入力 α, β, γ)
{
      // x はローカル変数

      if (α) {
        x = 1
      }
      if (β > 9) {
        if (! α && γ) {
            x = x+1
        }
        x = x+2
      }

      Return x
}
```

Returnに至るまでのパスを解析する。

それぞれのパスについての制約をSATソルバで判定することで、
パスを通すための入力データ α、β、γ の値を特定する



もくじ

- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

シンボリック実行からコンコリックテストへ

- これは革命的な技術である！これで全パス網羅のテストが自動化できるか？
- しかしながら、現実のソフトウェアへの適用には（他の技術と同様）様々な制約がある
- 制約を軽減し、より効率的により高いコードカバレッジを達成するためにシンボリック実行を基盤にした技術が**コンコリックテスト**である

現実のソフトウェアへの適用における様々な力べ

- パス数の爆発
  - 簡単なプログラムでも、ループを含むとパス数は爆発的に増える
    - 10回のループが２つ並んでいたら、 10 x 10 = 100 のパス
  - 対象ソフトウェアの規模が大きくなれば（いまだ）手に負えない
- ソルバの制約
  - ソルバの種類によって得手不得手あり
  - ビット演算、浮動小数点演算、ポインタ などなど
- オラクル問題
  - 入力値はわかった。で、出力が正しいかどうかどうやって判断するの？

コンコリックテスト（Concolic Test）とは？

- 具体的な値（Concrete value）を使ったプログラムの動的実行と、シンボリック実行（Symbolic execution）を組み合わせたもの
  - Concrete + Symbolic = Concolic
- 具体的な値をもとに制約を抽出し、ソルバで解く
  1. 入力となる変数（＝シンボルとして扱う変数）を指定
  2. 入力変数に任意の初期値を選択する
  3. プログラムを実行する
  4. 実行後に得られた「制約」式より、次に探索するパスを決定する。
     1. 一番最後に通った条件式の真理値を逆にした制約を作り、ソルバで具体的な入力データを特定する
     2. もしソルバで解けなかったら、さらに次の条件式の真理値を逆にして制約を作成し、ソルバを実行する
  5. ステップ４で得られた入力データをつかってステップ３へ

ステップ１： 入力をシンボルに置き換え

```
function(入力 a, b)
{
    // c はローカル変数

    c = b + 5

    if (a == c) {
      if ( a > 2 * b ) {
        return “fail”
      }
    }

    Return “pass”
}
```

シンボル a0, b0
で入力をおきかえ

```
function(入力 a0, b0)
{
    // c はローカル変数

    c = b0 + 5

    if (a0 == c) {
      if ( a0 > 2 * b0 ) {
        return “fail”
      }
    }

    Return “pass”
}
```

ステップ２：任意の初期値を入れてプログラムを実行

```
function(入力 a0, b0)
{
    // c はローカル変数

    c = b0 + 5

    if (a0 == c) {
      if ( a0 > 2 * b0 ) {
        return “fail”
      }
    }

    Return “pass”
}
```

a0 に 0, b0 に 0 を設定
※初期値はツールによって異なる

ステップ２：任意の初期値を入れてプログラムを実行

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 0, b0 に 0 を設定
※初期値はツールによって異なる

a0 = b0 + 5 は成り立たない（False）
プログラムは “pass” をリターンして終了

制約： a0 ≠ b0 + 5

ステップ３：条件判定の真偽が逆になるような入力の確定

```
function(入力 a0, b0)
{
      // c はローカル変数

      c = b0 + 5

      if (a0 == c) {
        if ( a0 > 2 * b0 ) {
           return “fail”
        }
      }

      Return “pass”
}
```

a0 に 0, b0 に 0 を設定
※初期値はツールによって異なる

a0 = b0 + 5 は成り立たない（False）
プログラムは “pass” をリターンして終了

制約：　a0 ≠ b0 + 5

True/Falseをひっくり返す

制約：　a0 = b0 + 5

ステップ３：条件判定の真偽が逆になるような入力の確定

```
function(入力 a0, b0)
{
      // c はローカル変数

      c = b0 + 5

      if (a0 == c) {
        if ( a0 > 2 * b0 ) {
           return “fail”
        }
      }

      Return “pass”
}
```

a0 に 0, b0 に 0 を設定
※初期値はツールによって異なる

a0 = b0 + 5 は成り立たない（False）
プログラムは “pass” をリターンして終了

制約：　a0 ≠ b0 + 5

True/Falseをひっくり返す

制約：　a0 = b0 + 5

SATソルバで制約が成り立つ値を確定
a0 : 10, b0 : 5

ステップ４：更新した入力値を入れてプログラムを実行

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 10, b0 に 5 を設定

ステップ４ ：更新した入力値を入れてプログラムを実行

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 10, b0 に 5 を設定

a0 = b0 + 5 が成り立つ（True）

制約： a0 = b0 + 5

ステップ４ ：更新した入力値を入れてプログラムを実行

```
function(入力 a0, b0)
{
      // c はローカル変数

      c = b0 + 5

      if (a0 == c) {
        if ( a0 > 2 * b0 ) {
           return “fail”
        }
      }

      Return “pass”
}
```

a0 に 10, b0 に 5 を設定

a0 = b0 + 5 が成り立つ（True）

制約： a0 = b0 + 5

a0 > 2 * b0 は成り立たない（False）
プログラムは”pass”をリターンして終了

制約2： a0 ≦ 2 * b0

ステップ５：条件判定の真偽が逆になるような入力の確定

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 10, b0 に 5 を設定

a0 = b0 + 5 が成り立つ（True）

制約： a0 = b0 + 5

a0 > 2 * b0 は成り立たない（False）
プログラムは”pass”をリターンして終了

制約2： a0 > 2 * b0

制約2： a0 ≦ 2 * b0

True/Falseを
ひっくり返す

ステップ５：条件判定の真偽が逆になるような入力の確定

```
function(入力 a0, b0)
{
    // c はローカル変数

    c = b0 + 5

    if
      i


    }

    Return “pass”
}
```

a0 に 10, b0 に 5 を設定

+ 5 が成り立つ（True）

制約： a0 = b0 + 5

SATソルバで2つの制約が
成り立つ値を確定
a0 : 7, b0 : 2

> 2 * b0 は成り立たない（False）

グラムは”pass”をリターンして終了

制約2： a0 > 2 * b0

制約2： a0 ≦ 2 * b0

True/Falseを
ひっくり返す

ステップ６：さらに更新した入力値を入れてプログラムを実行

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 7, b0 に 2 を設定

a0 = b0 + 5 が成り立つ（True）

制約：　a0 = b0 + 5

a0 > 2 * b0 が成り立つ（True）
プログラムは”fail”をリターンして終了

制約2：　a0 > 2 * b0

ステップ６：さらに更新した入力値を入れてプログラムを実行

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = b0 + 5

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

a0 に 7, b0 に 2 を設定

a0 = b0 + 5 が成り立つ（True）

制約：　a0 = b0 + 5

a0 > 2 * b0 が成り立つ（True）
プログラムは”fail”をリターンして終了

制約2：　a0 > 2 * b0

全パス網羅できた！

動的な実行の利点

```
function(入力 a0, b0)
{
     // c はローカル変数

     c = foo(b0)

     if (a0 == c) {
       if ( a0 > 2 * b0 ) {
          return “fail”
       }
     }

     Return “pass”
}
```

foo(x) = x+5

実行した結果を元に「制約」を定義するため、
問題を単純化できる＝実現可能性が高まる

もくじ

- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

コンコリックテストを実行するツール群

- PathCrawler
  - C言語に対応したツール。オンラインで試すことができる。
    - http://pathcrawler-online.com/
- CUTE/jCUTE
  - CおよびJava言語に対応したツール。最近更新がない。
    - https://github.com/osl/jcute
- CREST
  - C言語に対応したツール。国内でも活用事例あり。
    - http://jburnim.github.io/crest/
- KLEE
  - LLVM（Javaのバイトコードのようなもの）を利用したツール。LLVMに変換できる言語であれば対応可能。（C,C++,Objective C, Goなど）
    - https://klee.github.io/
- Microsoft Pex
  - Visual Studio 2010より機能が実装された.NET Framework向けのツール。

    2015バージョンでは、Smart Unit Testsという名前に進化。
    - https://msdn.microsoft.com/ja-jp/library/dn823749.aspx
- SPF (Symbolic Path Finder)
  - Java言語に対応したコンコリックテストの親戚ツール。目的は同じ。

    NASAでの適用事例あり。Eclipseプラグインがあり使いやすい。
    - http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
- DART,SAGE
  - Microsftの非公開ツール。Fuzzテストで活用されているらしい。

ツールによる処理の流れ（CRESTの例）

もくじ

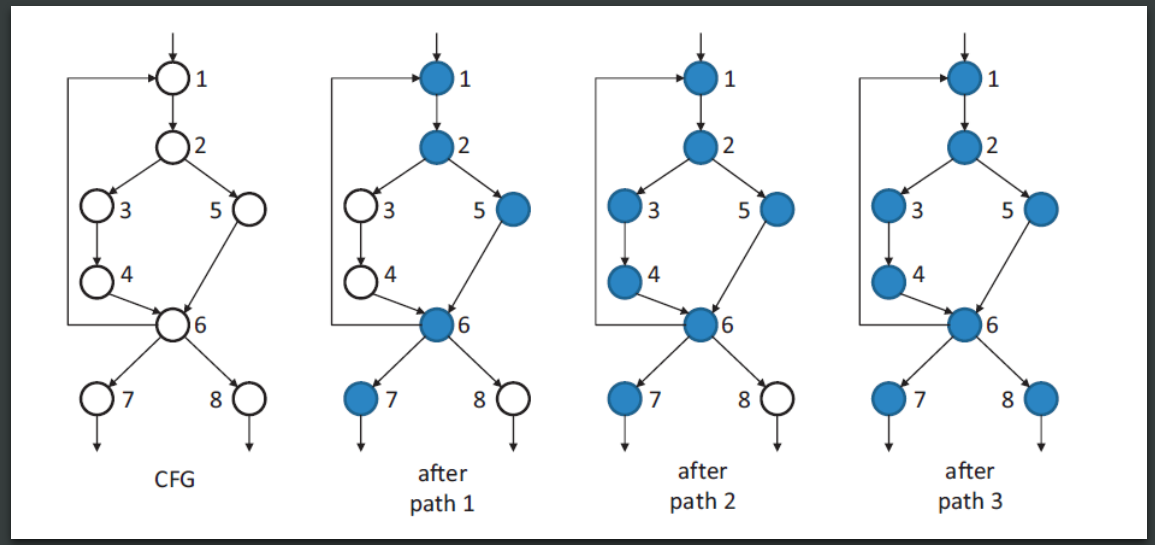
- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

実用例

- Fuzzテストに広く応用されている。従来のFuzzテストはブラックボックス（またはグレーボックス）のテストであったが、コンコリックテスト技術によってホワイトボックスで探索ができるようなった。
  - MicrosoftのSAGE（ツール）による事例が有名。
    - http://research.microsoft.com/en-us/um/people/pg/public_psfiles/SAGE-in-one-slide.pdf
- Samsungの組み込みソフトウェアへの適用事例
  - http://swtv.kaist.ac.kr/publications/2012/icst2012-industry.pdf
- デンソーの組み込みソフトウェア単体テストへの適用事例
  - http://www.sea.jp/ss2015/paper/ss2015_C1-4(2).pdf
- HP、IBMなどでも活用されているらしいが詳細は不明
  - https://en.wikipedia.org/wiki/Concolic_testing

コンコリックテストの課題

- シンボリック実行の課題は解決できたのか？
  - パス数の爆発
    - アルゴリズムの進化とコンピュータの処理性能により改善した
    - しかし課題は多くあり、部分的な適用や実行履歴を使った効率化などの工夫が必要



Path cutting technique

カバレッジレベルを
条件カバレッジに制約

参考: Tokumoto, S et al, “Enhancing Symbolic Execution to Test the Compatibility of Re-engineered Industrial Software Industrial Software”, Software Engineering Conference (APSEC), 2012 19th Asia-Pacific

コンコリックテストの課題

- シンボリック実行の課題は解決できたのか？
  - ソルバの制約
    - ソルバの進化によって改善
    - SMTソルバ（SATソルバの進化形）の進化は続いている
  - オラクル問題
    - 本質的な問題は変化なし
    - 後述する応用研究で解決案が提案されている

もくじ

- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

ツールのデモ

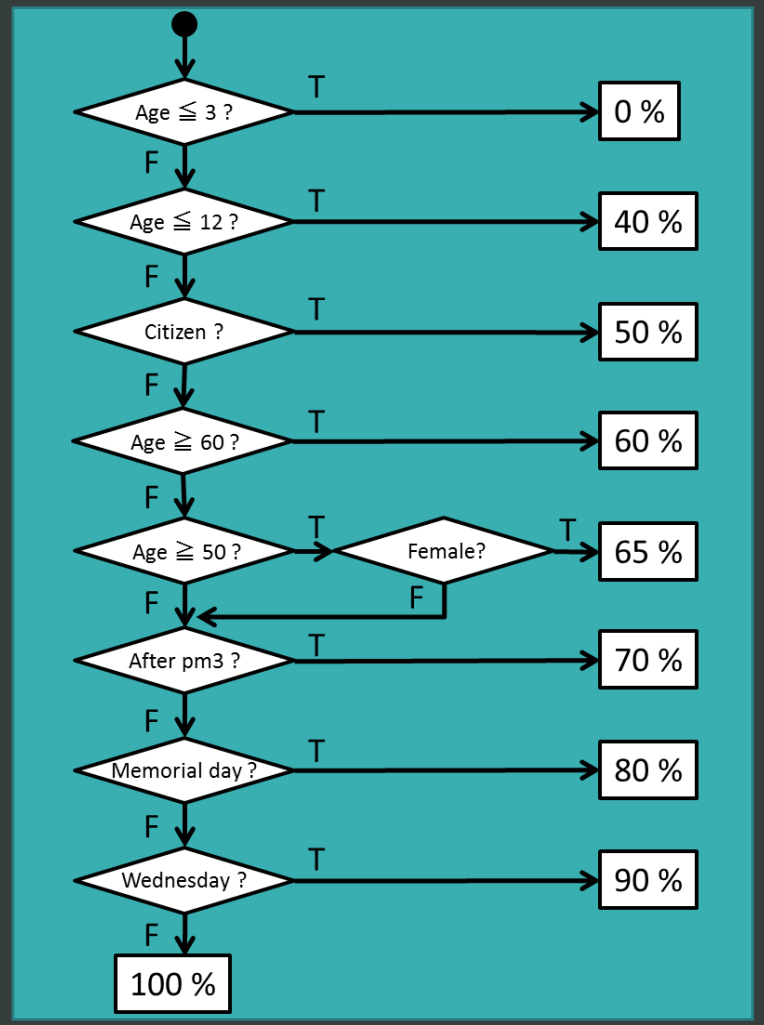
- ツール
  - Symbolic Path Finder
- インストールは割と簡単
  - 詳細手順はこちら
    - http://blog.utkeiji.com/?p=4

ある施設の入場割引の仕様

- 3歳以下の場合無料
- 水曜日なら正規料金の90%
- 60歳以上なら正規料金の60%
- 女性の場合、50歳以上なら正規料金の65%
- 記念日なら正規料金の80%
- 地域住民なら正規料金の50%
- 15時以降なら正規料金の70%
- 12歳以下なら正規料金の40%
- ただし条件が重複する場合は割引の大きい方を選択する

松尾谷、“Concolic Testing を活用した実装ベースの回帰テスト”, ソフトウェアシンポジウム2015 より抜粋

実装されたプログラムのフローチャート



もくじ

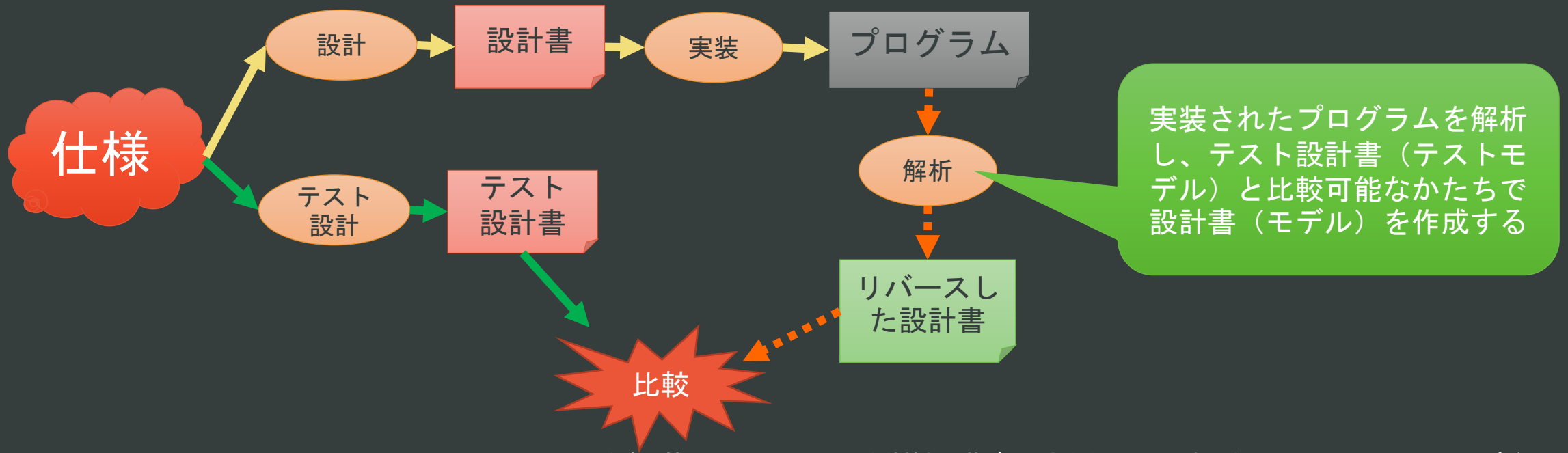
- 動的テストと静的テスト
- コンコリックテストの原理
  - 基盤技術：シンボリック実行、SATソルバ
  - コンコリックテストのアルゴリズム
- コンコリックテストを実行するツール
- 事例紹介
- デモ
- 応用技術

さまざまな応用可能性

- ソフトウェア開発中の活用
  - 決定表（デシジョンテーブル）による効率的な静的テスト
    - 参考：植月, “ソフトウェアの実装情報に基づく決定表を活用した論理検証手法”, ソフトウェアシンポジウム2013
  - 形式仕様からのテストケース自動生成
    - ※ アイデアレベル
- 派生開発・保守への活用
  - ソフトウェア修正の影響分析
    - 参考： 松尾谷、”Concolic Testing を活用した実装ベースの回帰テスト”, ソフトウェアシンポジウム2015
    - 参考： Keiji Uetsuki et al.,”Compatibility Testing Method for Software Logic by Using Symbolic Execution”, InSTA 2015

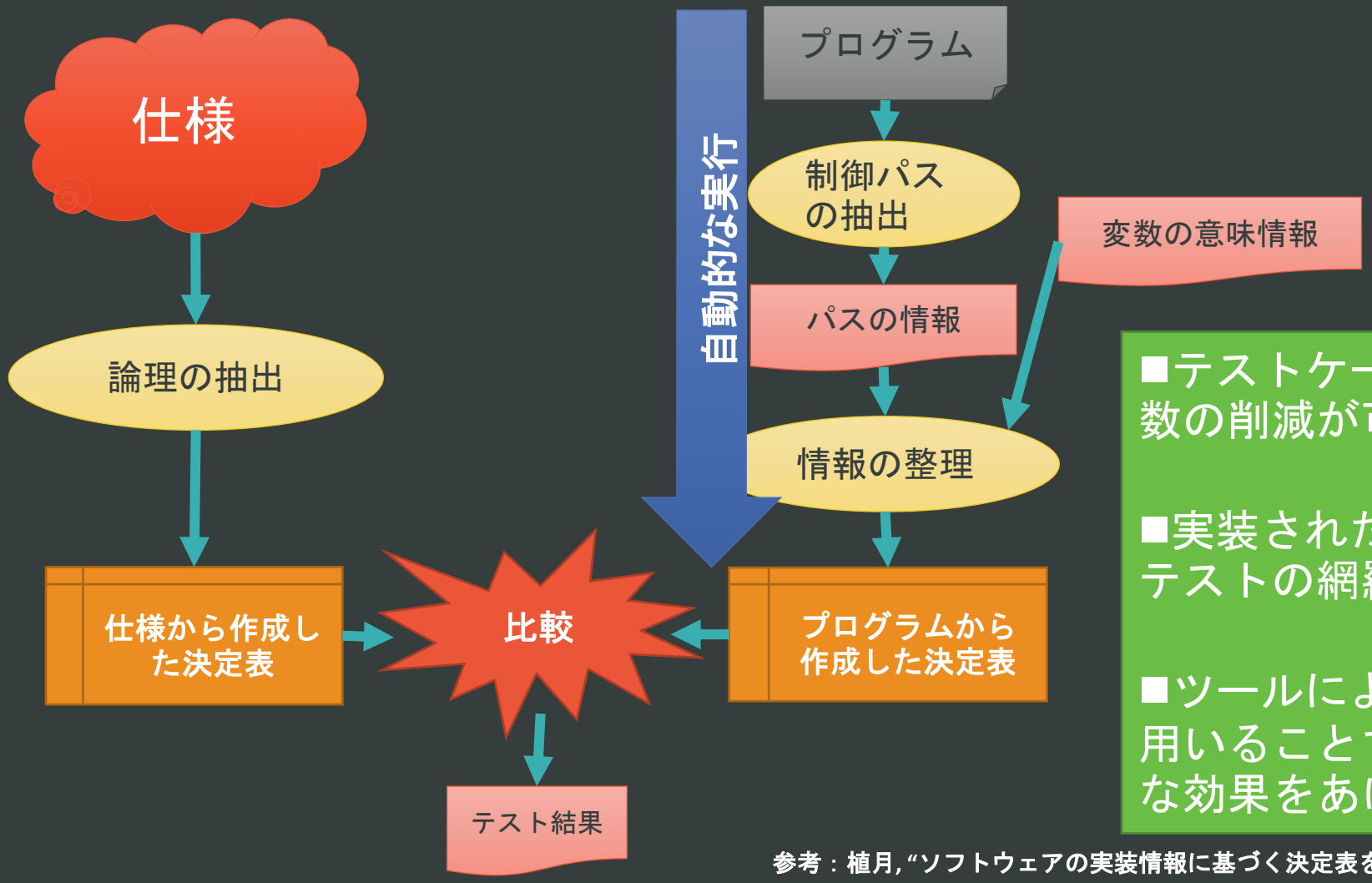
決定表（デシジョンテーブル）による効率的な静的テスト

- 実装されたソフトウェアから抽象度の1段高い情報（設計情報）を抽出し、テストのモデルと比較することでテスト効率化を達成する
- コンコリックテストによって「論理」の比較を行う
- 表現手段として決定表（デシジョンテーブル）を使う



参考：植月, “ソフトウェアの実装情報に基づく決定表を活用した論理検証手法”, ソフトウェアシンポジウム2013

決定表による論理検証の流れ



■テストケースの作成を行わないため作業工数の削減が可能である

■実装された論理がもれなく抽出できるためテストの網羅性が向上する

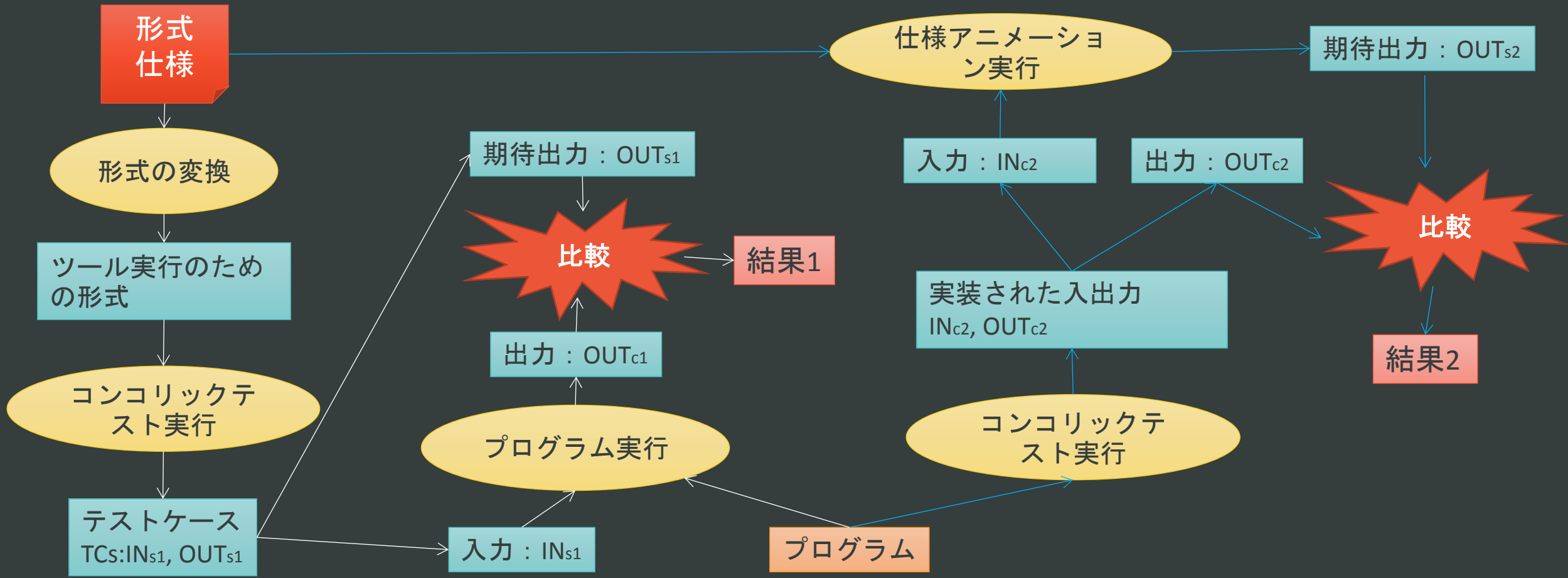
■ツールによる自動化および形式的な表現を用いることで作業の多様性をなくし、定量的な効果をあげられる

参考：植月, “ソフトウェアの実装情報に基づく決定表を活用した論理検証手法”, ソフトウェアシンポジウム2013

形式仕様からのテストケース自動生成

- 形式的仕様記述による仕様モデルに対して、コンコリックテストを適用してテストケースを生成する
  - 類似の先行研究あり
    - Shaoying Liu and Shin Nakajima, “A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications”, Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference
- しかしこれまでの方法では、仕様の間違い、実装漏れは検出できるが、仕様にない実装は検出できない
- そこでプログラムから同様に入出力の対を生成し、仕様との比較を行うことで完全なバグの検出をおこなう

形式仕様からのテストケース自動生成

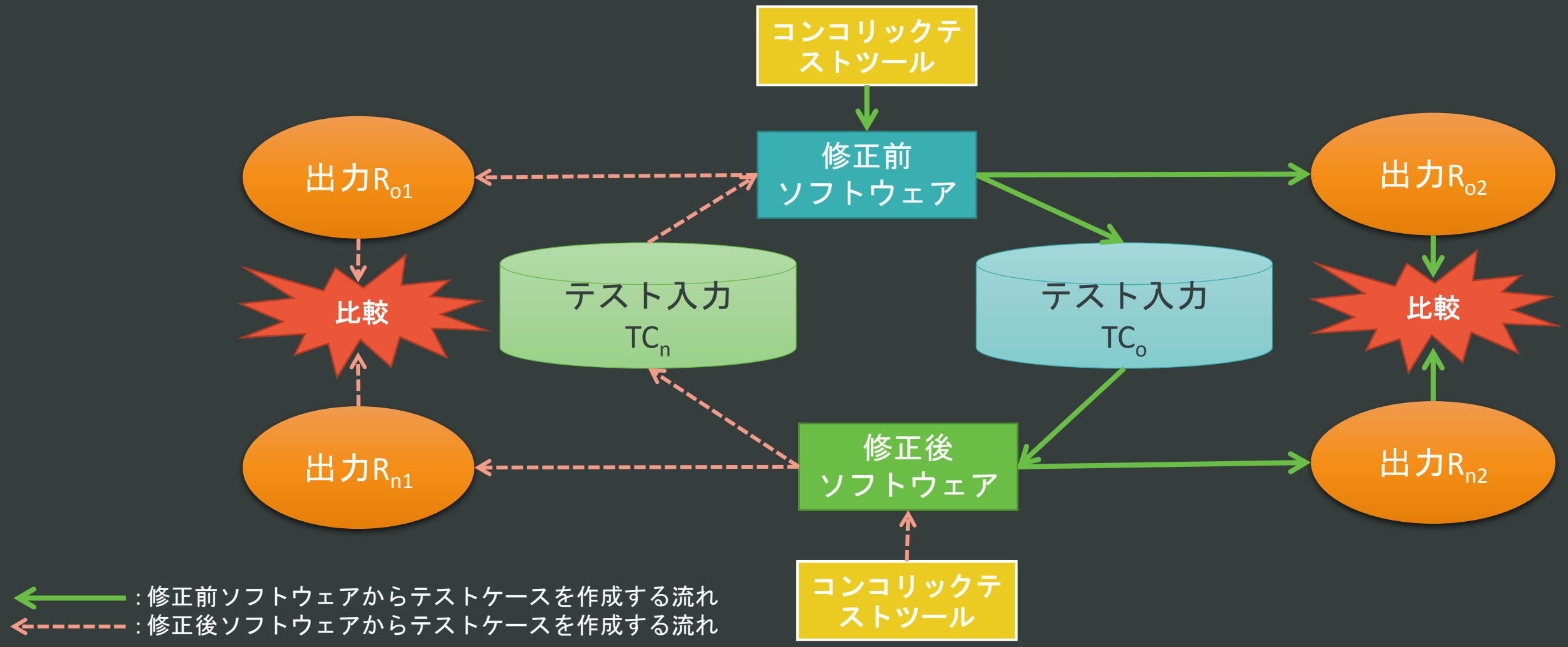


結果１によって、プログラムの仕様間違い、仕様未実装が検出できる
結果２によって、仕様にない実装が検出できる

ソフトウェア修正の影響分析

- 保守や派生開発において、元のソースコードに変更・追加をおこなったときの影響が把握できないために、デグレードの発生や、類似の機能をもつコードの追加などの課題がある
- コンコリックテストによって修正前後のプログラムの入出力を明らかにすることで、その差分から影響範囲の妥当性を検証する
  - 静的な情報（制御フロー、データフロー）の範囲
  - 処理時間や割り込みタイミングの変化などは範囲外

ソフトウェア変更による影響を確認するプロセス



参考： Keiji Uetsuki et al.,"Compatibility Testing Method for Software Logic by Using Symbolic Execution", InSTA 2015

まとめ

- アルゴリズムの研究、コンピュータ処理能力の向上により、コードの静的解析技術は飛躍的に進歩している
  - ソフトウェア開発のシンギュラリティ（技術的特異点）は近い？
- コンコリックテスト（シンボリック実行）はソフトウェアテストの改善に有望な技術である
- 様々な言語に対応したツールがそろってきている
- 一方で、実践への適用には制約があり、工夫して使う必要がある。しかしながらそのノウハウが不十分な状況である
  - とくに日本ではほとんどない

ぜひみなさんの現場で実践してみてください。そしてノウハウを共有し、日本のソフトウェア開発の現場をハッピーにしていきましょう！