

A Review of Software Change Impact Analysis



Steffen Lehnert

Technical Report, Ilmenau University of Technology, Department of Software Systems /
Process Informatics, Ilmenau, Germany, December, 2011.

紹介者
宮崎 智己

内容

- ◆ Change Impact Analysis（影響波及分析）の
サーベイ論文
 - ▶ 全182件(1978年～2011年)の論文をサーベイ
 - ▶ 手法や目的別にまとめている
- ◆ 膨大な数になるためちょっとずつ紹介していきます

影響波及分析

- ◆ 変更による影響, または変更するために必要な修正を特定すること
 - ▶ “Identify the potential consequences of a change, or estimating what needs to be modified to accomplish a change”[1]
- ◆ 何に役立つか？
 1. 変更に必要なタスクの量がわかる
 2. 変更すべき箇所を提案できる
 3. 再実行すべきテストケースの認識を助ける

[1] S. A. Bohner and R. S. Arnold, Software Change Impact Analysis. Los Alamitos, CA, USA: IEEE Computer Society Publications Tutorial Series, 1996.

影響波及分析が必要な理由

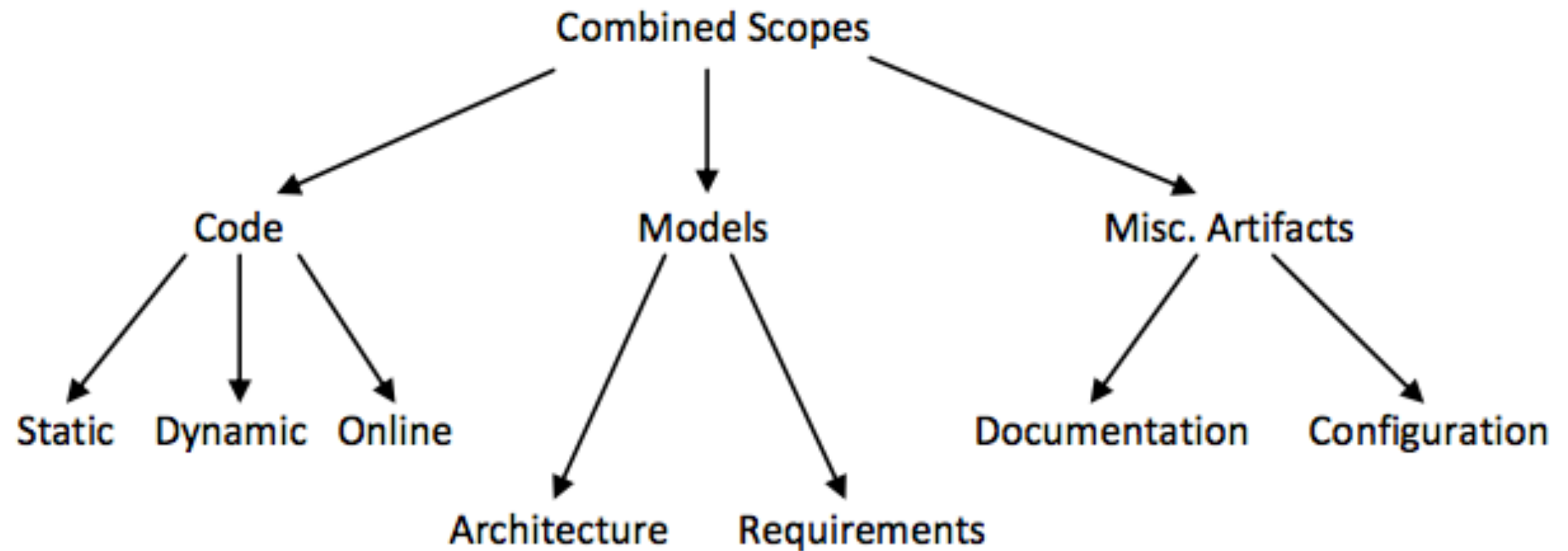
- ◆ メンテナンスは非常にコストがかかる
 - ▶ メンテナンスコストの50%以上はソフトウェアの変更によるもの
- ◆ 部分的な変更は高いリスクとなる
 - ▶ 予期しない副作用
 - ▶ 新しいバグの混入
 - ▶ 変更する前よりも不安定な状況
- ◆ 変更によって生じる影響をきちんと把握すべき

影響波及分析の分析対象

1. ソースコード
2. アーキテクチャモデル
 - ・ 主にUMLコンポーネント図
3. 要件定義書モデル
4. アーティファクト
 - ・ ドキュメント
 - ・ 設定ファイル
5. ごちゃ混ぜ

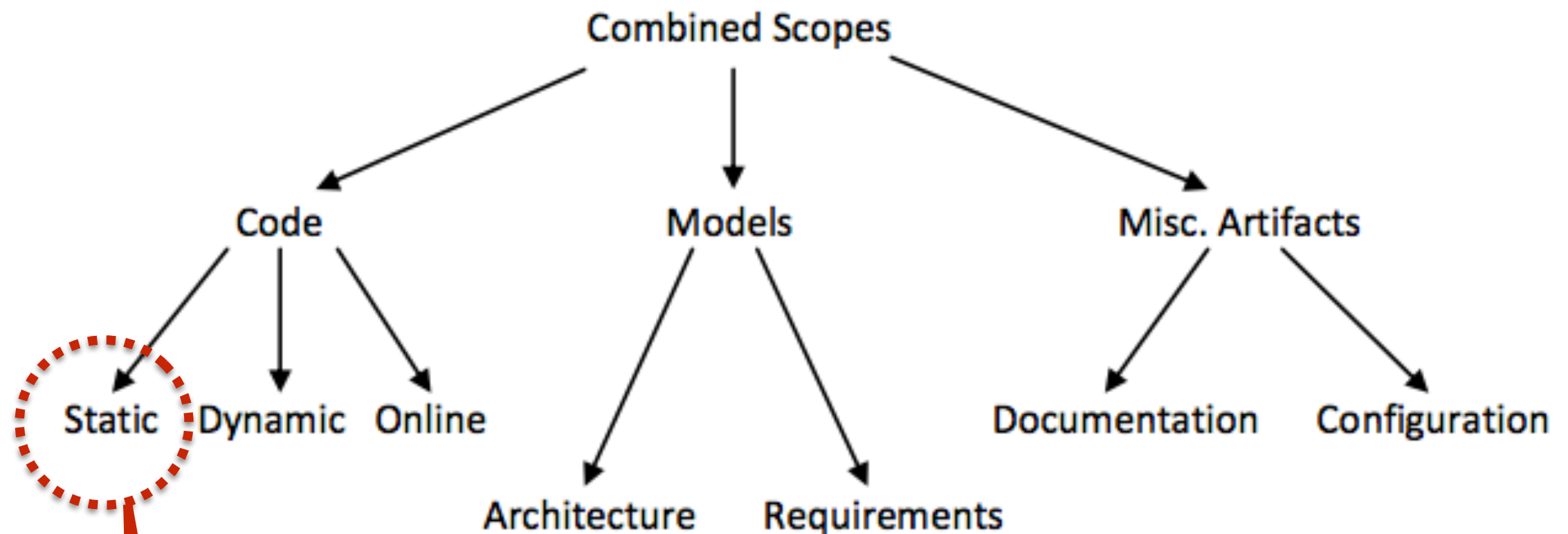
影響波及分析の分類

.....



影響波及分析の分類

.....



まずは静的解析から話していきます

静的解析(Static code Analysis)

- ◆ コードを実行せずに（静的に）構造や動作を解析する方法
- ◆ 解析できるもの
 - ▶ 欠陥の検出
 - ▶ 脆弱性の検出：Ex).SQLインジェクション
 - ▶ コードメトリクスの計測：Ex).行数, 複雑度
 - ▶ OSSのコード混入の検出

静的解析の利点・欠点

◆ 利点

- ▶ コーディング時に行える
- ▶ 早い段階でエラーを検出可
- ▶ コードの可読性や保守性の向上

◆ 欠点

- ▶ コードの実行順序やパスに起因する問題は検出不可
- ▶ 例外, メモリ破壊などの実行時も問題の検出は厳しい

静的解析の主な手法

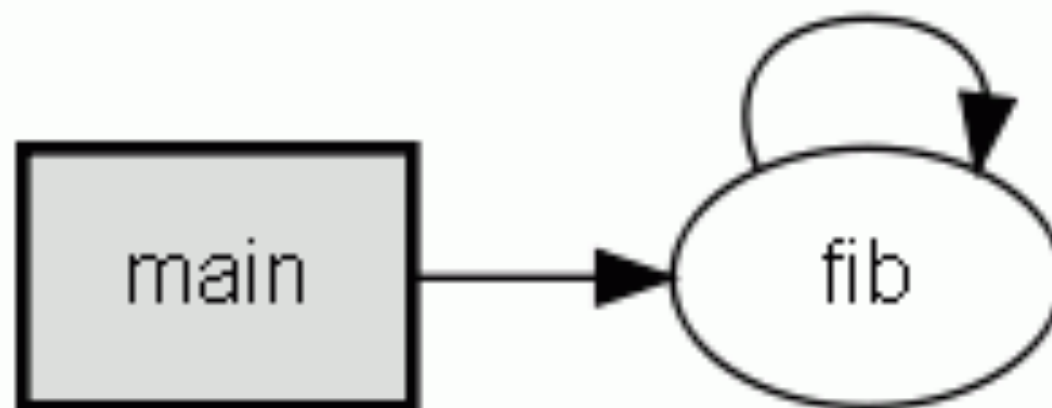
- ◆ Call graph ←今日はこれ
- ◆ Dependency Analysis
- ◆ Program Slicing

Call graph

- ◆ メソッド呼出の関係を表現した有向グラフ
 - ▶ エッジは (f, g) で表される : f が g を呼び出す

```
1  #include <stdio.h>
2
3  int fib(int n)
4  {
5      if (n<3) return 1;
6      return fib(n-2)+fib(n-1);
7  }
8
9  int main()
10 {
11     int i;
12     for (i=1; i<=10; ++i) {
13         printf("fibonacci(%d) = %d\n", i, fib(i));
14     }
15     return 0;
16 }
```

メソッド
呼び出し



エッジ

- $(\text{main}, \text{fib})$
- (fib, fib)



Ryderらの手法[2]

- ◆ テストの失敗の原因となった変更を特定
 - ▶ Eclipseのプラグインとして実装
- ◆ 手順
 1. ソースを解析し, 相互依存関係のatomic changesを取得
 2. オリジナルのプログラムのコールグラフと対照して, 取得した変更と関連のあるテストケースのコールグラフを作成
 3. オリジナルのプログラムのcall graphと関連づけて, テストに影響を与えている変更を決定

atomic changes

.....

(AC)	Add an empty class
(DC)	Delete an empty class
(AM)	Add an empty method
(DM)	Delete an empty method
(CM)	Change body of method
(LC)	Change virtual method lookup
(AF)	Add a field
(DF)	Delete a field

Table 1: Categories of atomic changes.

Lookupの定義

.....

$Lookup = \{ \langle C, A.m, B.m \rangle \mid \begin{array}{l} \text{class } A \text{ contains virtual method } m, C \leq^* B \leq^* A, \\ \text{class } B \text{ contains virtual method } m, \\ \text{there is no class } B' \text{ that contains method } m \text{ such that } C \leq^* B' <^* B \end{array} \}$

$LC = \{ \langle a, b \rangle \mid \langle a, b, c \rangle \in ((Lookup_{old} - Lookup_{new}) \cup (Lookup_{new} - Lookup_{old})) \}$

A, B, C : クラス

$Lookup_{old}$: 変更前のLookup

$Lookup_{new}$: 変更後のLookup

$A < B$: AはBの子クラス

$A \leq B$: AはBの子クラス, またはA=B

$A \leq^* B$: BはAの親クラス, またはA=B

$A <^* B$: BはAの親クラス

影響分析の公式

$$\begin{aligned} \text{AffectedTests}(\mathcal{T}, \mathcal{A}) &= \{ t_i \mid t_i \in \mathcal{T}, \text{Nodes}(P, t_i) \cap (\mathbf{CM} \cup \mathbf{DM}) \neq \emptyset \} \cup \\ &\quad \{ t_i \mid t_i \in \mathcal{T}, n \in \text{Nodes}(P, t_i), n \rightarrow_B A.m \in \text{Edges}(P, t_i), \langle B, X.m \rangle \in \mathbf{LC}, B <^* A \leq^* X \} \\ \text{AffectingChanges}(t, \mathcal{A}) &= \{ a' \mid a \in \text{Nodes}(P', t) \cap (\mathbf{CM} \cup \mathbf{AM}), a' \preceq^* a \} \cup \\ &\quad \{ a' \mid a \equiv \langle B, X.m \rangle \in \mathbf{LC}, B <^* A \leq^* X, n \rightarrow_B A.m \in \text{Edges}(P', t), \\ &\quad \text{for some } n, A.m \in \text{Nodes}(P', t), a' \preceq^* a \} \end{aligned}$$

$\text{AffectTests}(T, A)$: 変更Aによって起きたかもしれないテストドライバのTのサブセット

$\text{AffectingChanges}(t, A)$: テストtに影響を及ぼしたかもしれない変更Aのサブセット

$\text{Node}(P, t_i)$: プログラムPのメソッド

$\text{Edge}(P, t_i)$: メソッド間の呼出関係

$\text{Node}(P', t_i)$: P' (Pの変更後) のメソッド

$\text{Edge}(P', t_i)$: メソッド間の呼出関係

Xiaらの手法[3]

- ◆ 変更はデータと制御部分に影響を与えると考えられるが、データの変更にのみ着目
 - ▶ 名前, 型, スコープ, etc
 - ▶ 代入演算子を含まない制御部分には影響をそれほど与えないから
- ◆ 影響の大きさを変数dの変更によって影響を受けたコンポーネントMのLOCで判断

[3]F. Xia and P. Srikanth, "A change impact dependency measure for predicting the maintainability of source code," in Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04), vol. 2, September 2004, pp. 22–23.

影響波及予測の式

$$C_{predicted} = \alpha \cdot \left[\sum_{i=0}^{\infty} \beta^i b_i \right]$$

$$b_i = \begin{cases} \text{LOC of M} & i = 1 \\ \text{LOC of subprograms immediately called in M} & i > 1 \end{cases}$$

α : 実際に影響を受けるor受けないを分類する係数

β^i : b_i の減衰係数

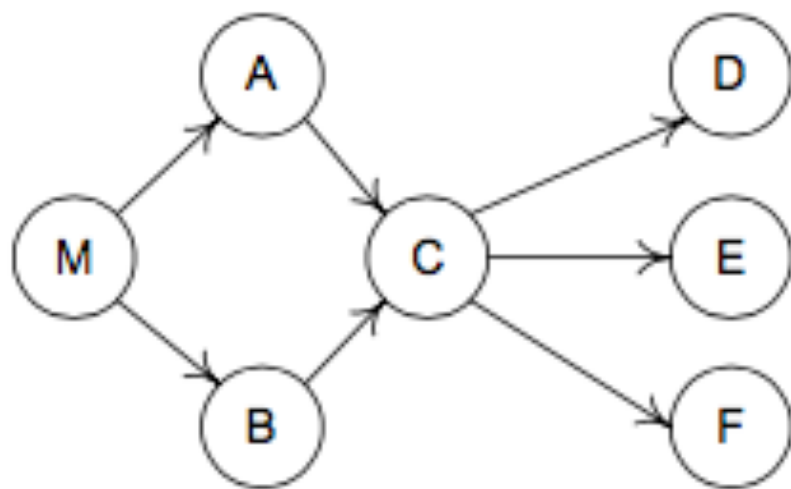
α と β は Δk_i の平均二乗誤差が最小になる値
($\alpha \doteq 1.00$, $\beta \doteq 0.90$)

$$\Delta_k = \alpha \cdot \left[b_0 + \sum_{i=1}^{\infty} \beta^i \cdot b_i \right] - C_{Actual}(k)$$

Badriらの手法[4]

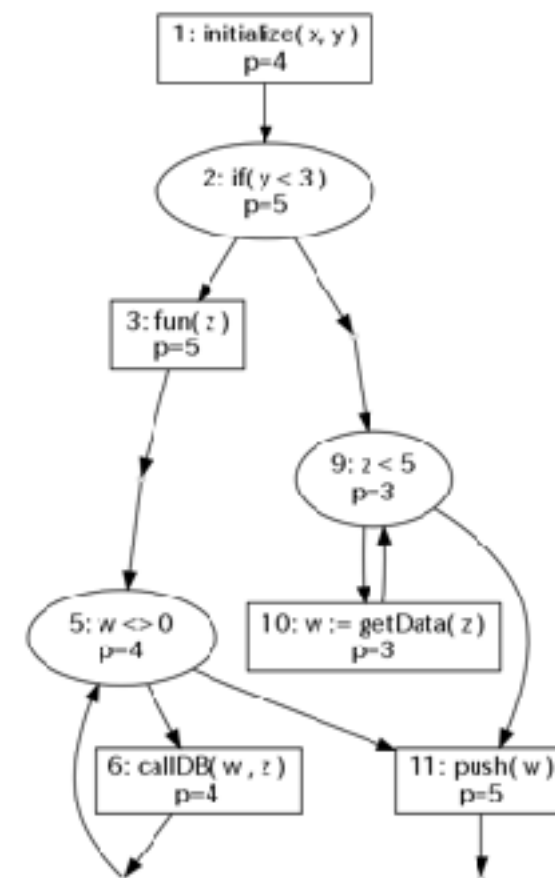
◆ Control call graphの提案

- ▶ Call Graphに制御フロー情報を追加
- ▶ 処理の順序を考慮したCall graphに



Call graph

+

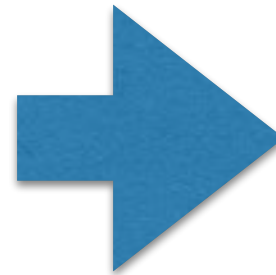


制御フローグラフ

Call graphだと

```
M()  
{  
  IF cond0 Then S1;  
    Else S2;  
  
  S3;  
  M1();  
  If cond1 Then M2();  
    Else M3();  
  While cond2 Do  
  {  
    M4();  
    If cond3 Then S4;  
    M5();  
  }  
  M6();  
}
```

ソースコード



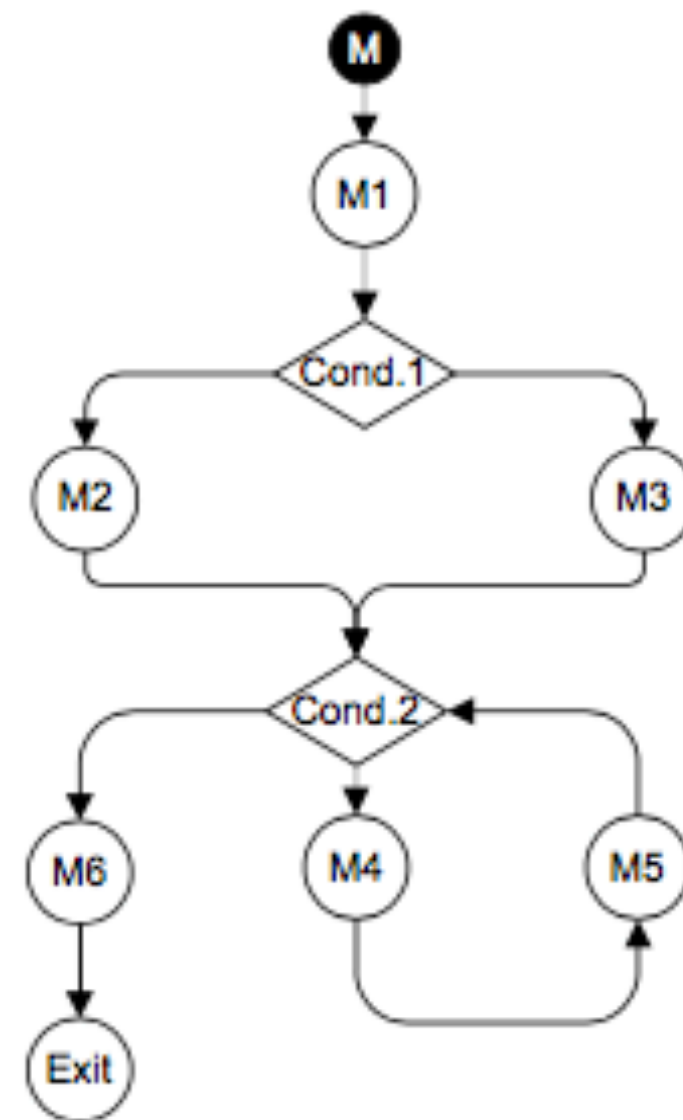
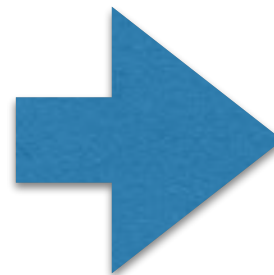
Call graph

余分な部分も計算してしまう

Control call graph

.....

```
M0
{
  IF cond0 Then S1;
    Else S2;
  S3;
  M1();
  If cond1 Then M2();
    Else M3();
  While cond2 Do
  {
    M4();
    If cond3 Then S4;
    M5();
  }
  M6();
}
```



ソースコード

Control Call graph

起こりえない実行経路を無視できる

まとめと所感

◆ まとめ

- ▶ 影響波及分析手法の一つに静的Call Graphがある
- ▶ Call Graphはメソッドの呼出関係を示したもの

◆ 所感

- ▶ 他の手法と比べるとそんなに数はなかった
 - ・ Dependency analysisは非常に多い