



A SURVEY OF SYMBOLIC EXECUTION TECHNIQUES

ROBERTO BALDONI ET.AL.
CSUR'2018

2018/12/11, 18, 25

0. ABOUT THIS PAPER

- 主にSymbolic Executionについての解説紙

- ConcolicやBackwardにも触れている

- 構成:

- 1章: 導入（表記法の定義）、簡単な例題、チャレンジ紹介
 - 2章: 記号実行エンジン、Concolic、Path Selection
 - 3章: メモリについて（チャレンジ1）
 - 4章: 環境について（チャレンジ2）
 - 5章: パス爆発について（チャレンジ3）
 - 6章: 制約解決について（チャレンジ4）
 - 7章: 考察

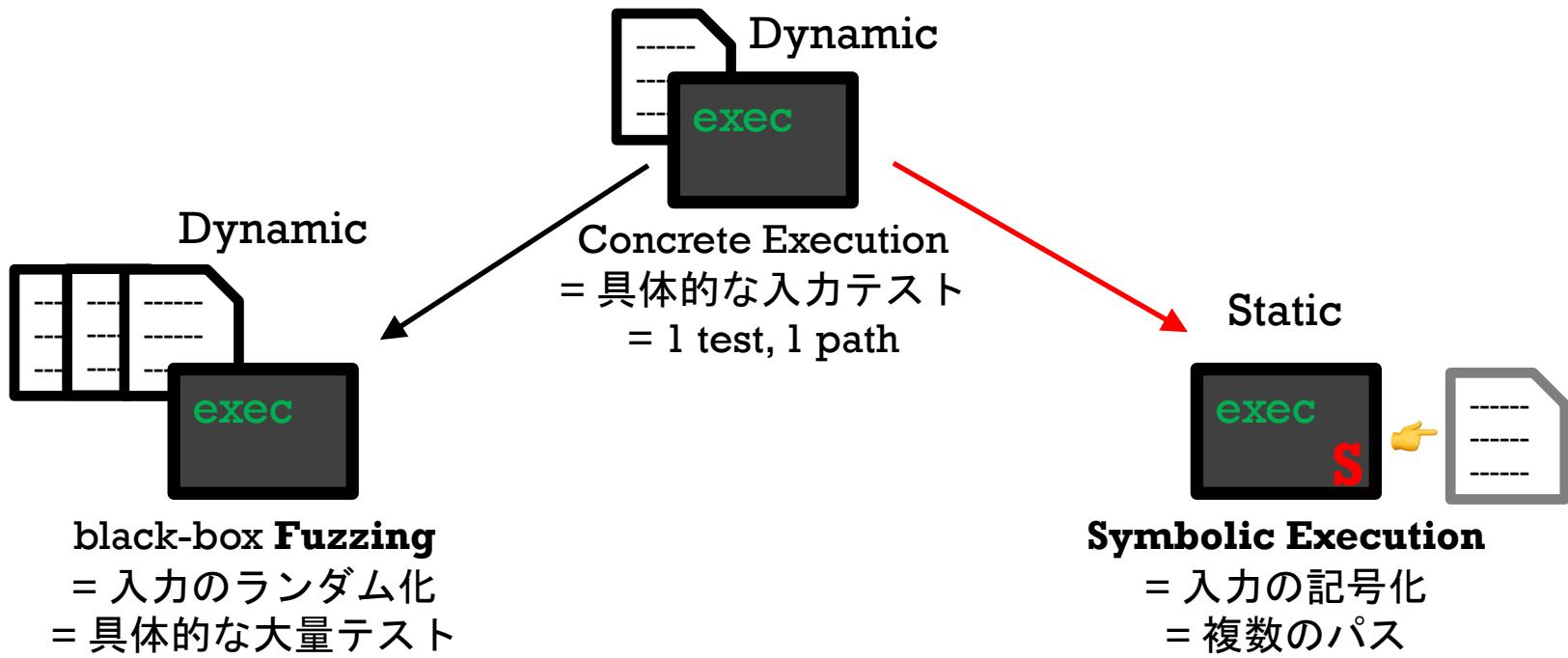
12/11

12/18

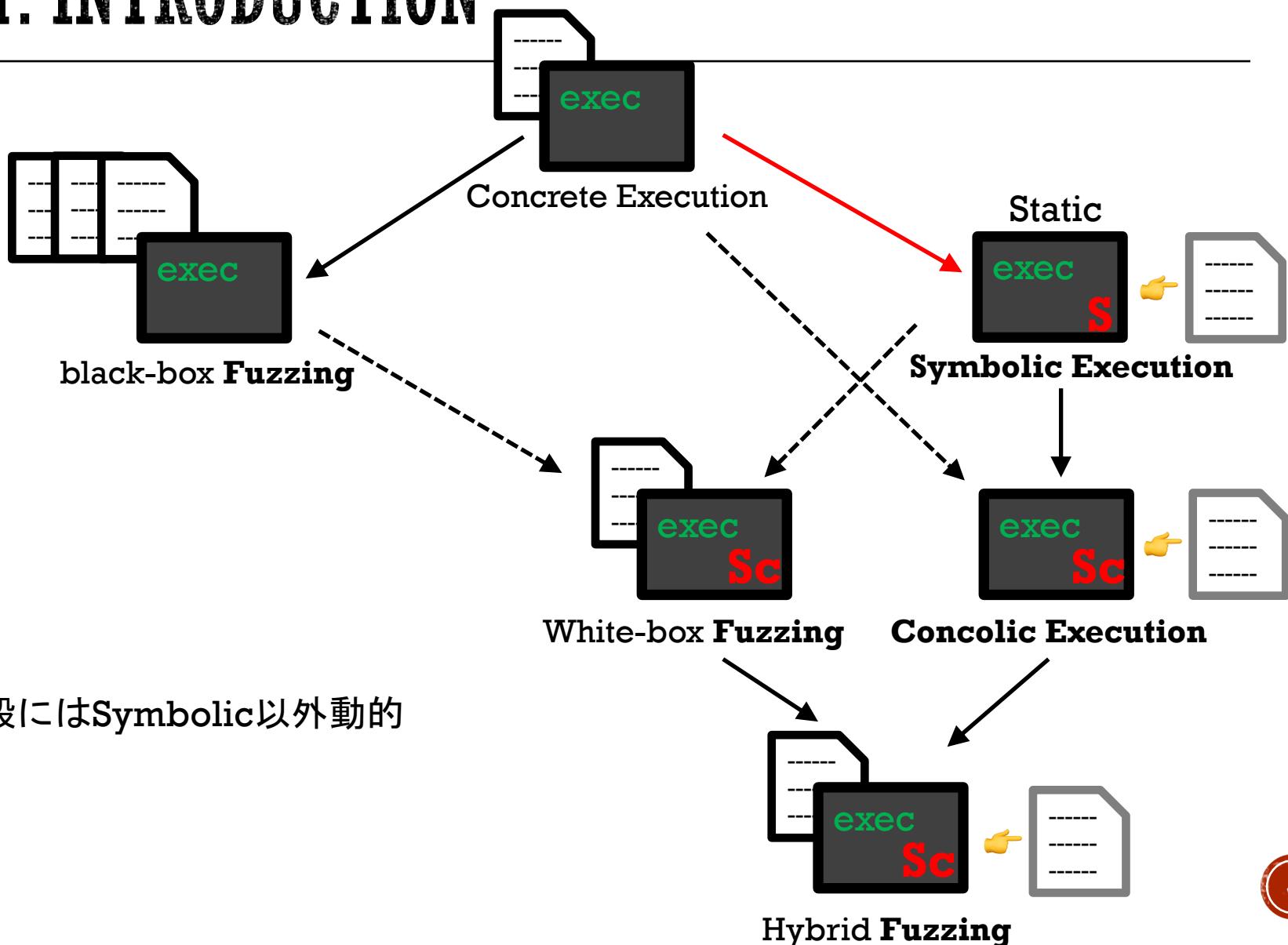
12/25

1. INTRODUCTION

- Symbolic Execution(記号実行)の背景
 - アイデア自体は70年代から存在 (0除算、ヌルポなどが対象)
 - 「大量テスト」 & 「脆弱性検出の自動化」が主目的
 - 同様の目的を持つ技術 → Fuzzing

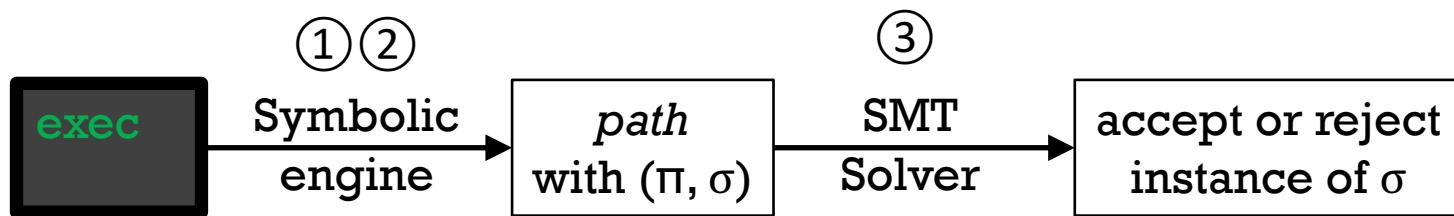


1. INTRODUCTION



1. INTRODUCTION

- Symbolic Executionのおおまかな流れ
 - ①入力を記号化し、**Symbolic Engine**上でシミュレーション
 - ②分岐条件(=制約)とシンボリックストアを各パスに関連付け
 - シンボリックストア: 変数と具体値or記号値の紐付け (後述)
 - ③パスの制約が満たされるかSMTソルバで検証
 - 加えてインスタンス(=条件を満たす具体的な入力例)の生成



1. INTRODUCTION 表記について

- 定義: Symbolic Engineの状態S
 - **S = (stmt, σ, π)**
 - stmt: 評価する次の文 (代入, 条件分岐, ジャンプ)
 - 関数呼び出し、loopは5章で解説
 - σ: シンボリックストア (変数とその中身の対応関係)
 - 中身は具体値(concrete value) or 記号値式 (symbolic value: α_i)
 - π: パス制約
 - 初期値はTrue
 - それ以外は $\prod formula(\alpha_i)$ (= 記号値を持つ制約の積)
 - 理論上エンジン内では状態Sによって各パスは一意に定まる
 - マージとかアドレスの概念が入るとややこしくなるけど

1. INTRODUCTION 表記について

- $S = (\text{stmt}, \sigma, \pi)$
- stmtによる状態Sの更新
 - 代入 $x = e$:
 - $\sigma \rightarrow \sigma'$, $\text{stmt} \rightarrow \text{stmt} + 1$
 - σ' includes $x \rightarrow e_s$ (本来は|→みたいな矢印)
 - e_s : 実行状態で分けられた代入元の式e、単項or二項
 - 条件分岐 **if e stmt_true else stmt_false** :
 - $\pi \rightarrow \pi'$, $\text{stmt} \rightarrow \text{stmt_next}$
 - $\pi' = \pi \wedge e_s$ ($\text{stmt_next} == \text{stmt_true}$)
 - $\pi' = \pi \wedge \neg e_s$ ($\text{stmt_next} == \text{stmt_false}$)
 - e_s : boolean e を評価して得られる記号式
 - ジャンプ **goto stmt_target**:
 - $\text{stmt} \rightarrow \text{stmt_target}$

1. INTRODUCTION 例題: SOURCE CODE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい
 - 要するに $x-y == 0$ となるようなa, bは何じゃろな

```
1. void foobar(int a, int b) {  
2.     int x = 1, y = 0;  
3.     if (a != 0) {  
4.         y = 3+x;  
5.         if (b == 0)  
6.             x = 2*(a+b);  
7.     }  
8.     assert(x-y != 0);  
9. }
```

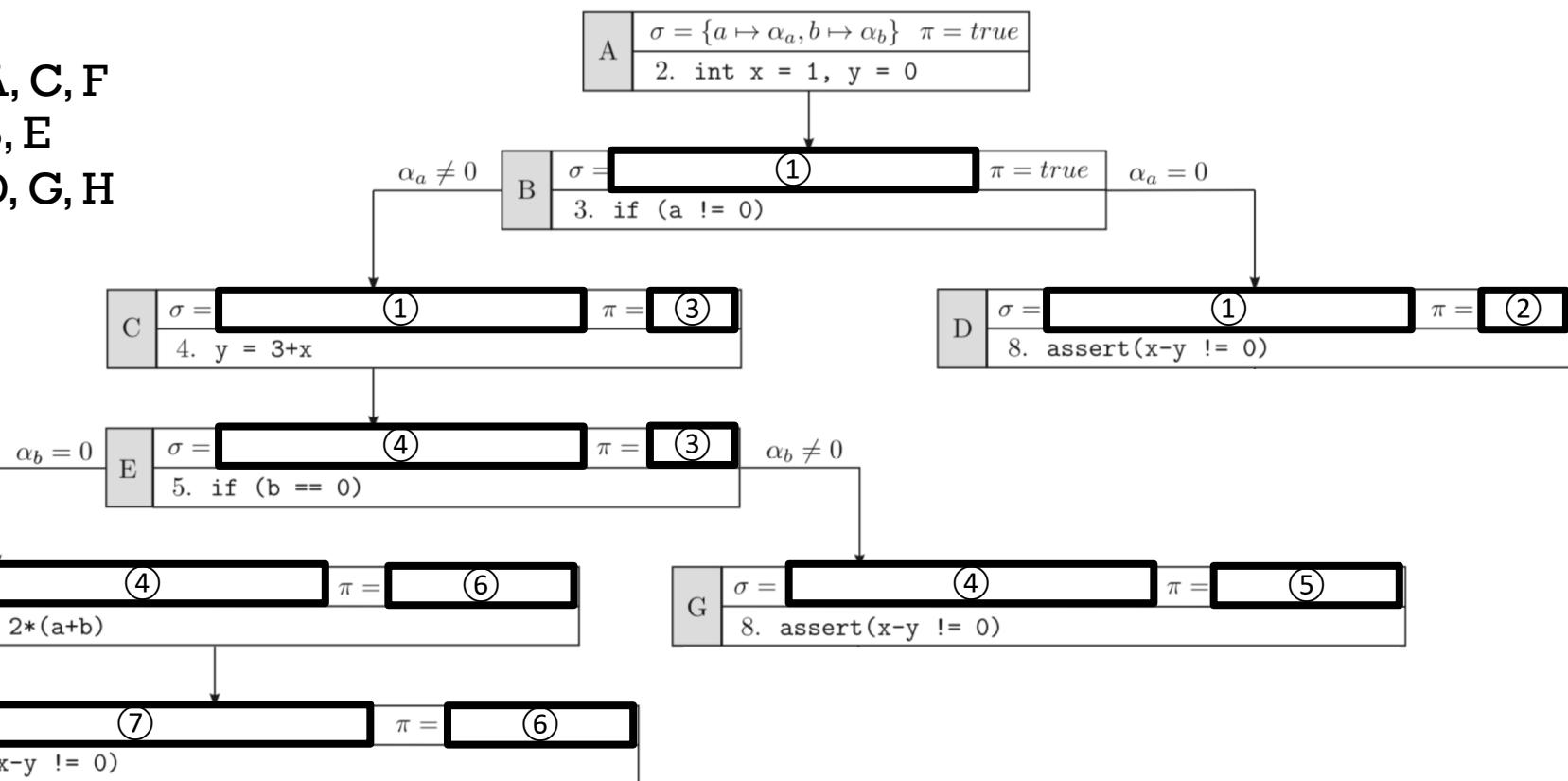
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



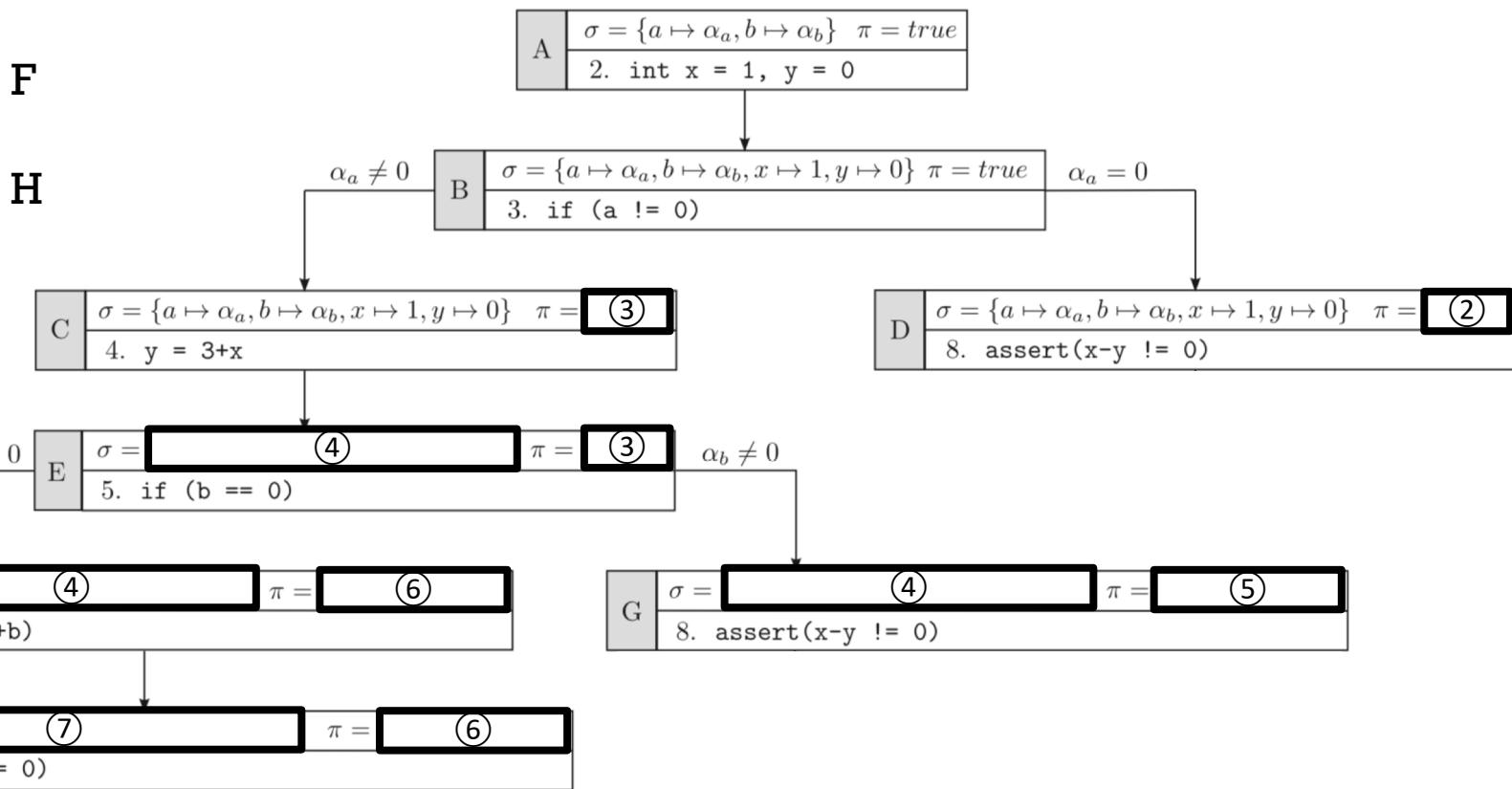
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



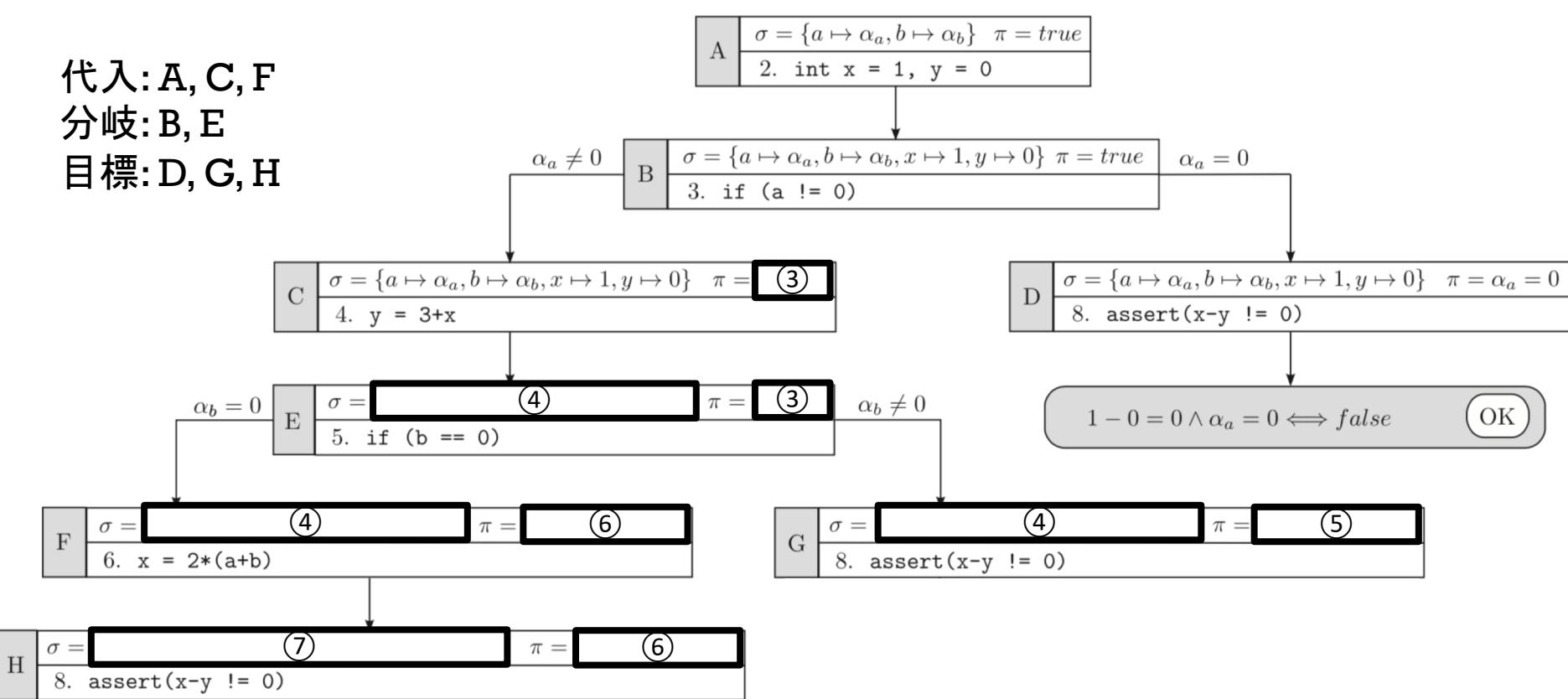
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



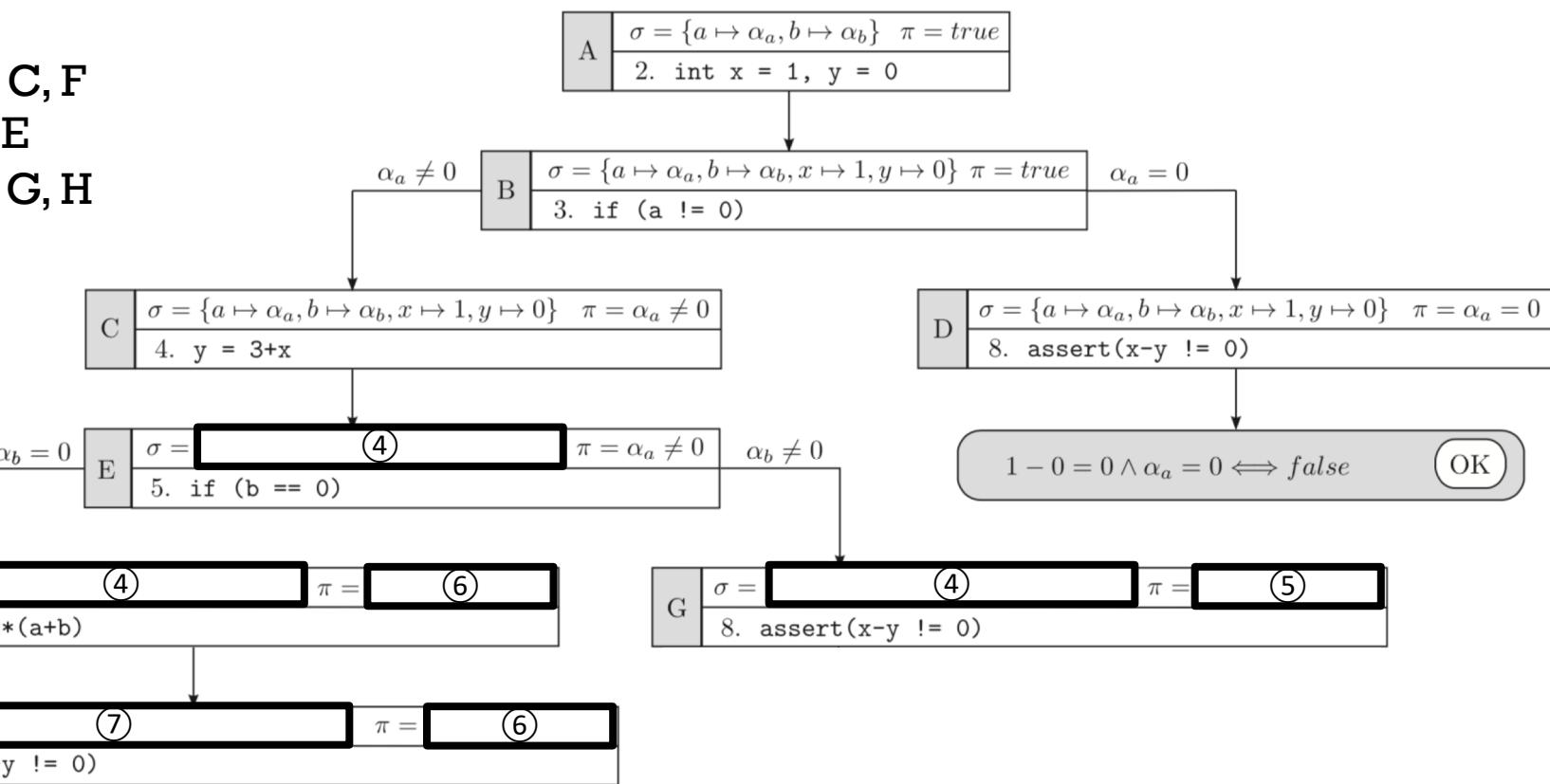
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



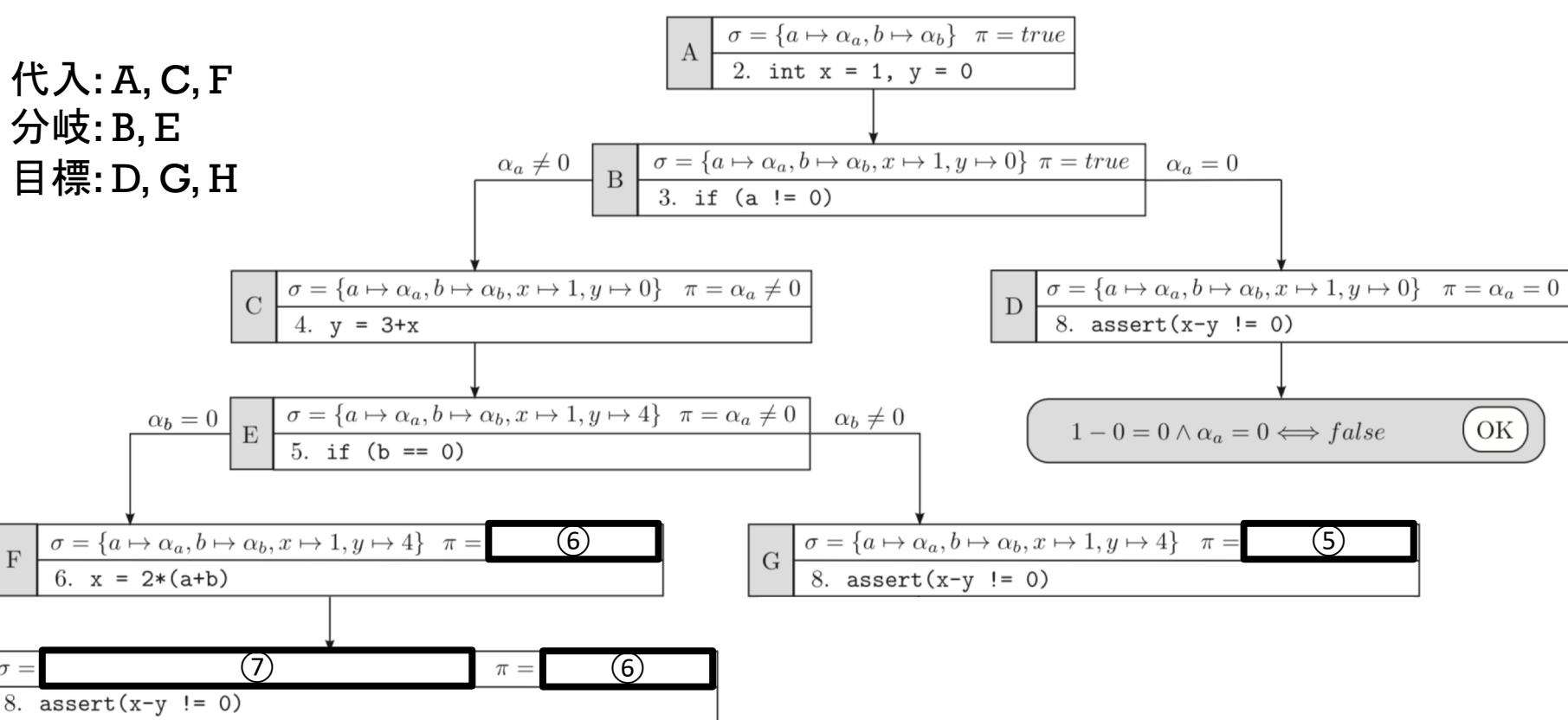
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



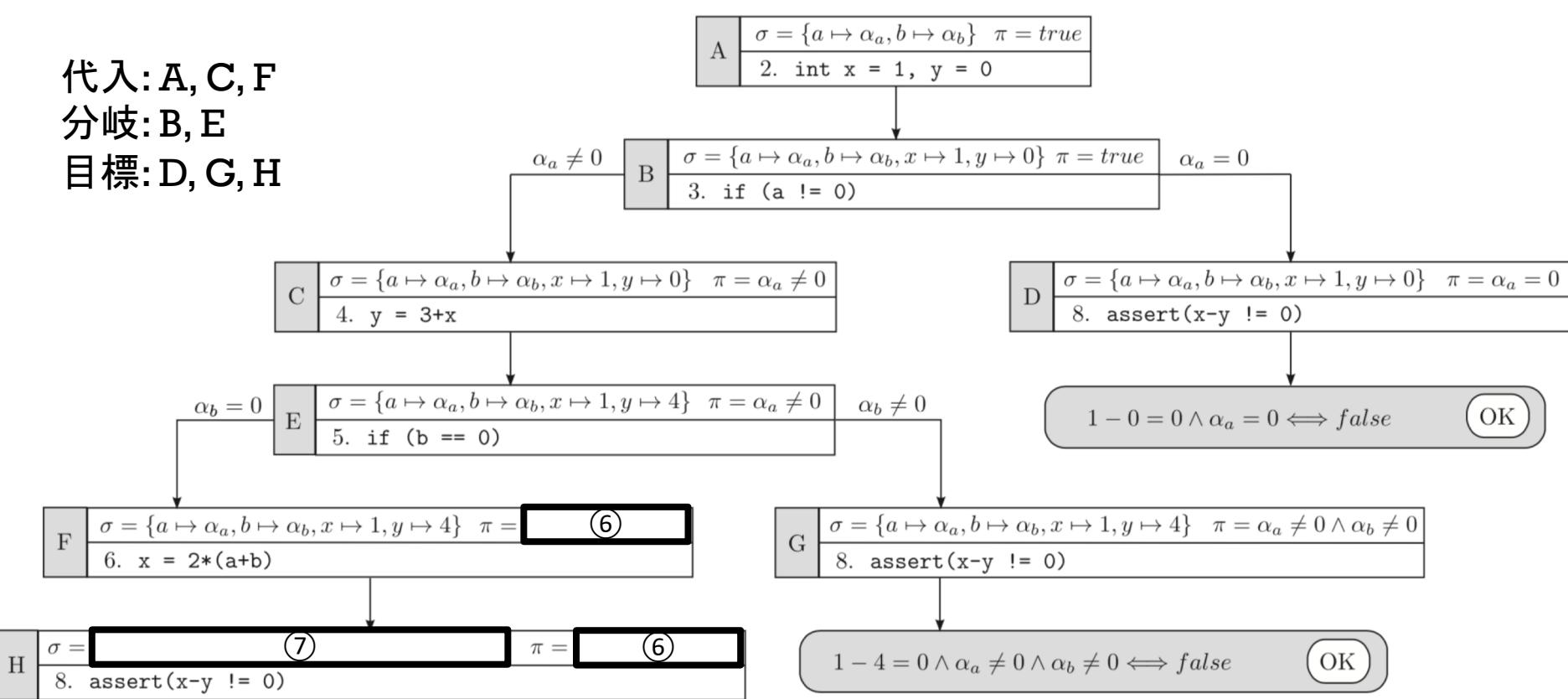
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



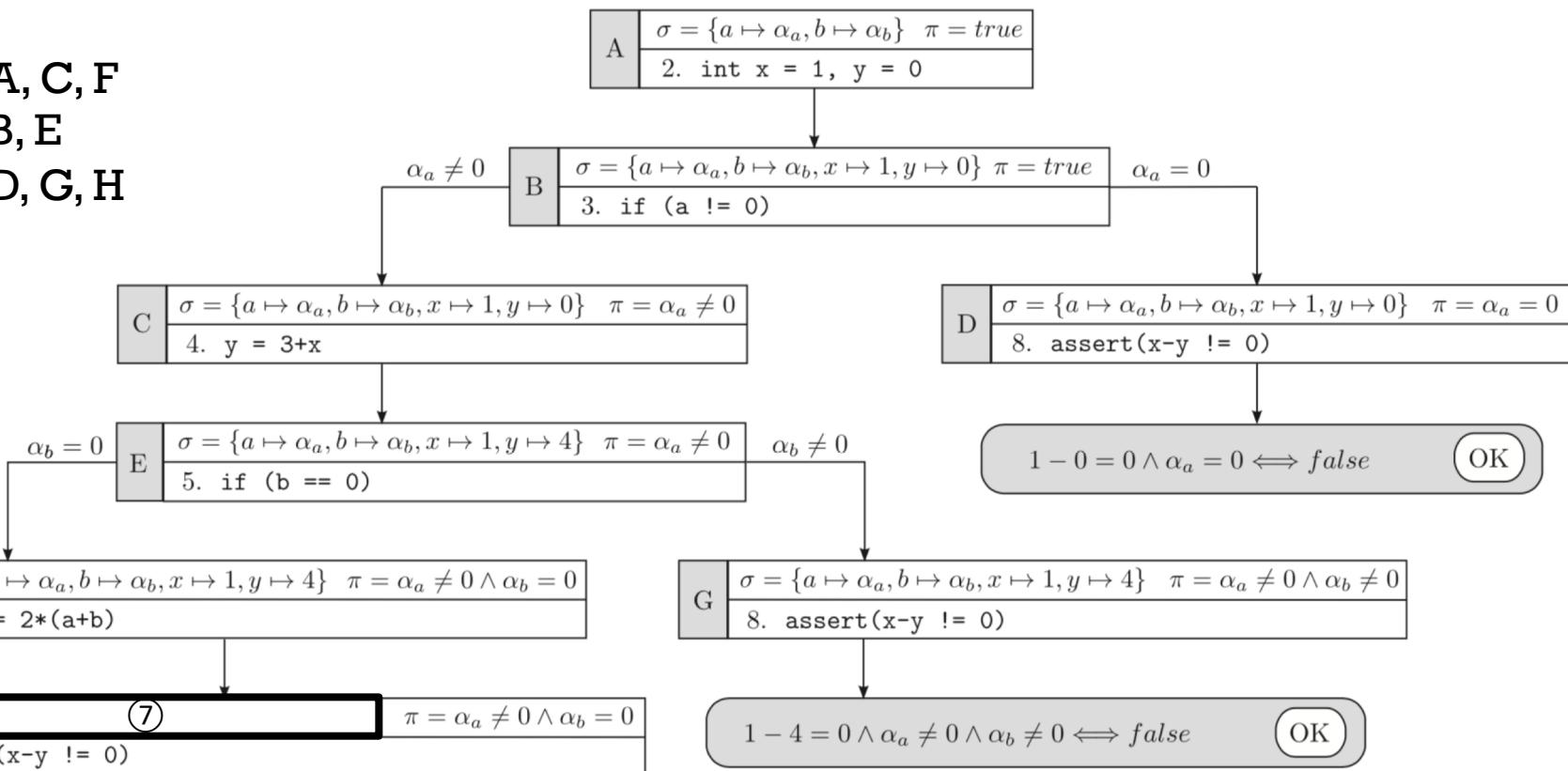
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



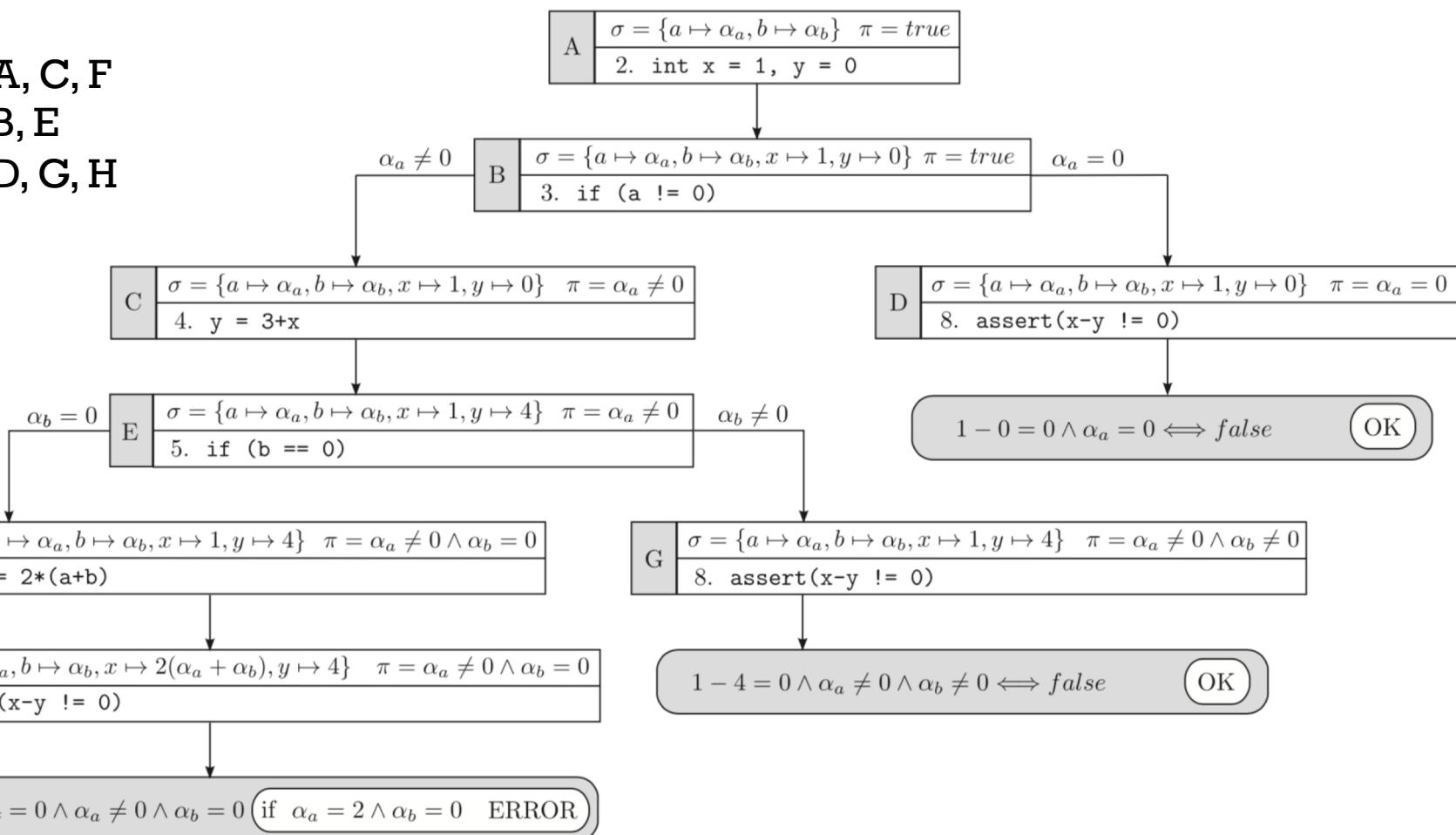
1. INTRODUCTION 例題: PATH-TREE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい

代入: A, C, F

分岐: B, E

目標: D, G, H



1. INTRODUCTION CHALLENGES

- Symbolic Executionの課題
 - 全パス網羅すれば健全性/完全性が保証されるが、パス爆発が発生
 - 健全性: 報告されなかった入力は確実に脆弱性を引き起こさない、を満たす
 - 完全性: 報告された入力は確実に脆弱性を引き起こす、を満たす
 - →大抵は健全性とパフォーマンスをトレードオフにする
- Symbolic Executionの*複雑な*課題
 - メモリ（3章）：
 - ポインタ、配列、その他オブジェクトの扱い（アドレスの概念）
 - 環境（4章）：
 - ソフトウェアスタック、システムコール、ファイルなどの扱い
 - 状態空間爆発（5章）：
 - パス爆発をどう抑えるか、ループなどの扱い
 - 制約の解決（6章）：
 - SMTソルバで解けない or 解くのに時間がかかる制約の扱い

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION

- Concolic = Concrete + Symbolic
 - Symbolic Executionは理論上全パスを探索可能
 - ただし現実にはスケールしない
 - 古典的な(≒ Static) Symbolic Executionの制限
 - 実行可能パスを探索しきれない
 - 非線形関数、超越関数が出るとお手上げ
 - 想定しきれない要素
 - 外部環境: ファイル、メモリ
 - スタック (静的に決定できない)



Dynamic Symbolic Execution

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

▪ Dynamic Symbolic Execution (DSE)

/ Dynamic Test Generation [51]

- エンジンの状態
 - $Sc = (\text{stmt}, \sigma, \sigma_c, \pi)$
 - σ_c : コンクリートストア、記号値に関連付けられた具体値
- Symbolic Executionと同様に制約(π)は収集
- 同時に実行も行う
 - 現在実行されているパスは存在することがSMTソルバなしで分かる
 - 何故なら実際に実行できているから
- 次の実行（パス）の選択:
 - 現在の実行制約の内、どこかの条件をトグル
 - →SMTソルバで検証、インスタンスの生成

[51] DART: Directed automated random testing [PLDI'05]

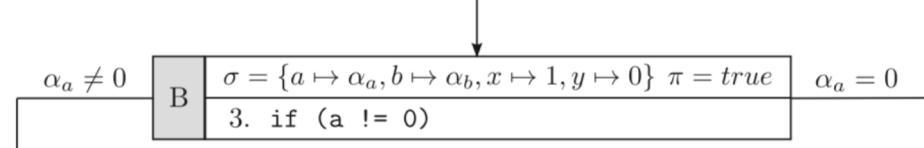
2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- 例題アゲイン: $\sigma_c(A)$ を求めてみよう (入力は $a = 1, b = 1$ と仮定)

$$\sigma_c(A) = \boxed{⑧}$$

A	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\} \quad \pi = \text{true}$
	2. int x = 1, y = 0



C	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\} \quad \pi = \alpha_a \neq 0$
	4. $y = 3+x$

$$\sigma_c(B, C) = \boxed{⑨}$$

E	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\} \quad \pi = \alpha_a \neq 0$
	5. if ($b == 0$)

$$\sigma_c(E, G) = \boxed{⑩}$$

G	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\} \quad \pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
	8. assert($x-y \neq 0$)

$$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \iff \text{false}$$

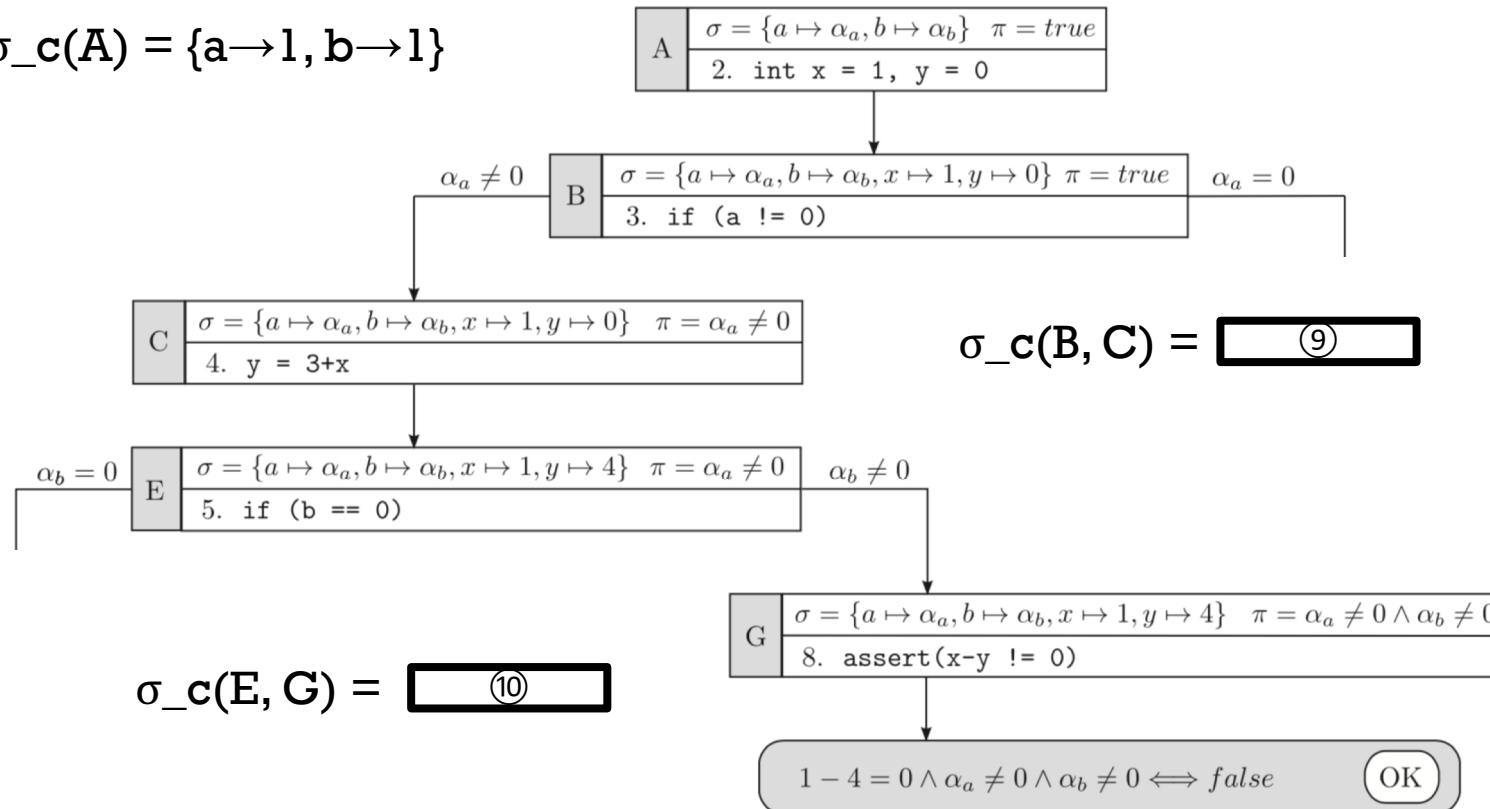
OK

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- 例題アゲイン: $\sigma_c(A)$ を求めてみよう (入力は $a = 1, b = 1$ と仮定)

$$\sigma_c(A) = \{a \rightarrow 1, b \rightarrow 1\}$$



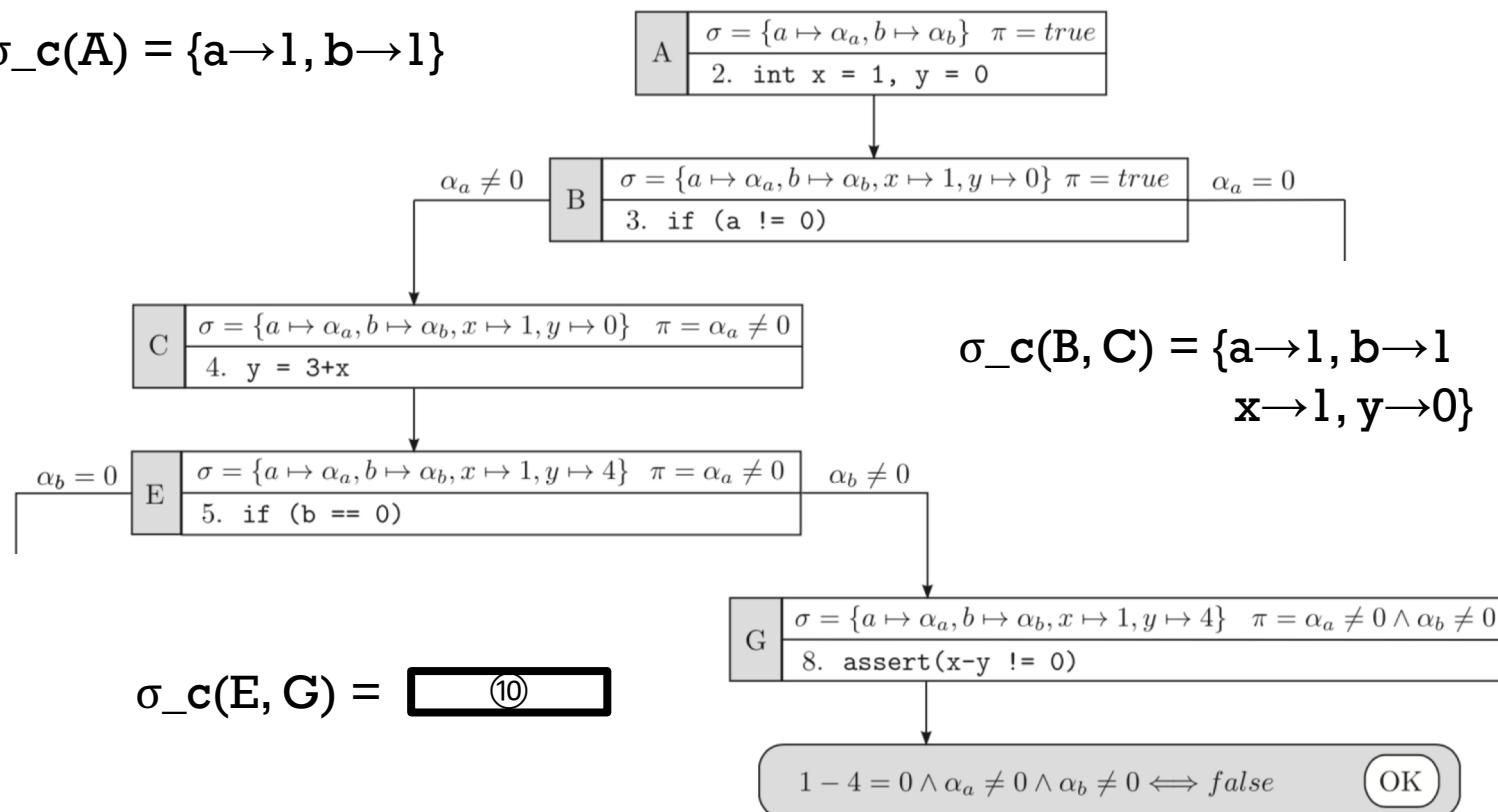
$$\sigma_c(E, G) = \boxed{⑩}$$

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- 例題アゲイン: σ_c を求めてみよう (入力は $a = 1, b = 1$ と仮定)

$$\sigma_c(A) = \{a \rightarrow 1, b \rightarrow 1\}$$



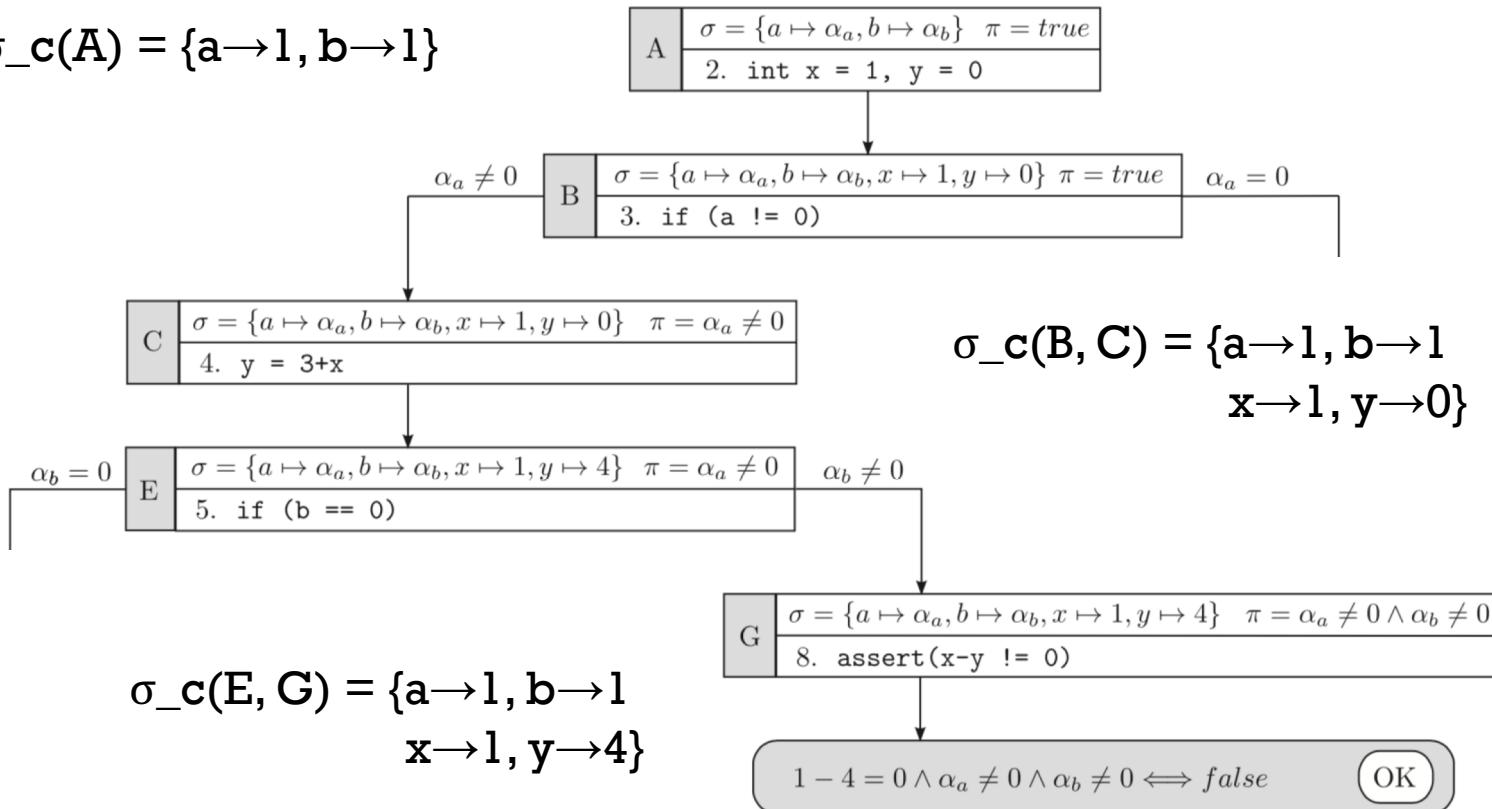
$$\sigma_c(E, G) = \boxed{10}$$

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- 例題アゲイン: $\sigma_c(A)$ を求めてみよう (入力は $a = 1, b = 1$ と仮定)

$$\sigma_c(A) = \{a \rightarrow 1, b \rightarrow 1\}$$



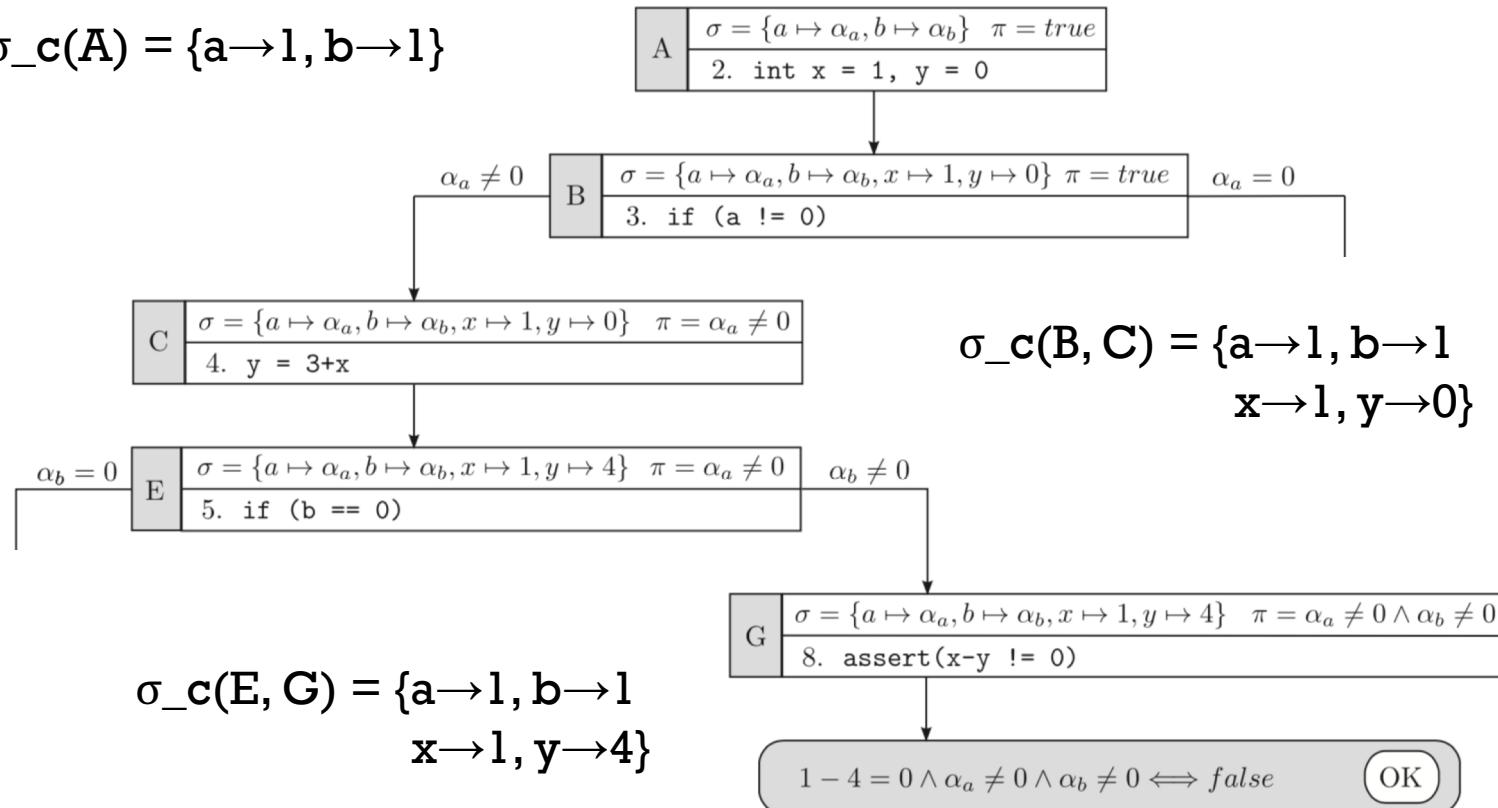
$$\sigma_c(E, G) = \{a \rightarrow 1, b \rightarrow 1, x \rightarrow 1, y \rightarrow 4\}$$

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- 例題アゲイン: σ_c を求めてみよう (入力は $a = 1, b = 1$ と仮定)

$$\sigma_c(A) = \{a \rightarrow 1, b \rightarrow 1\}$$



重要なこと

具体的な実行 σ_c と 記号実行 σ を併用している

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

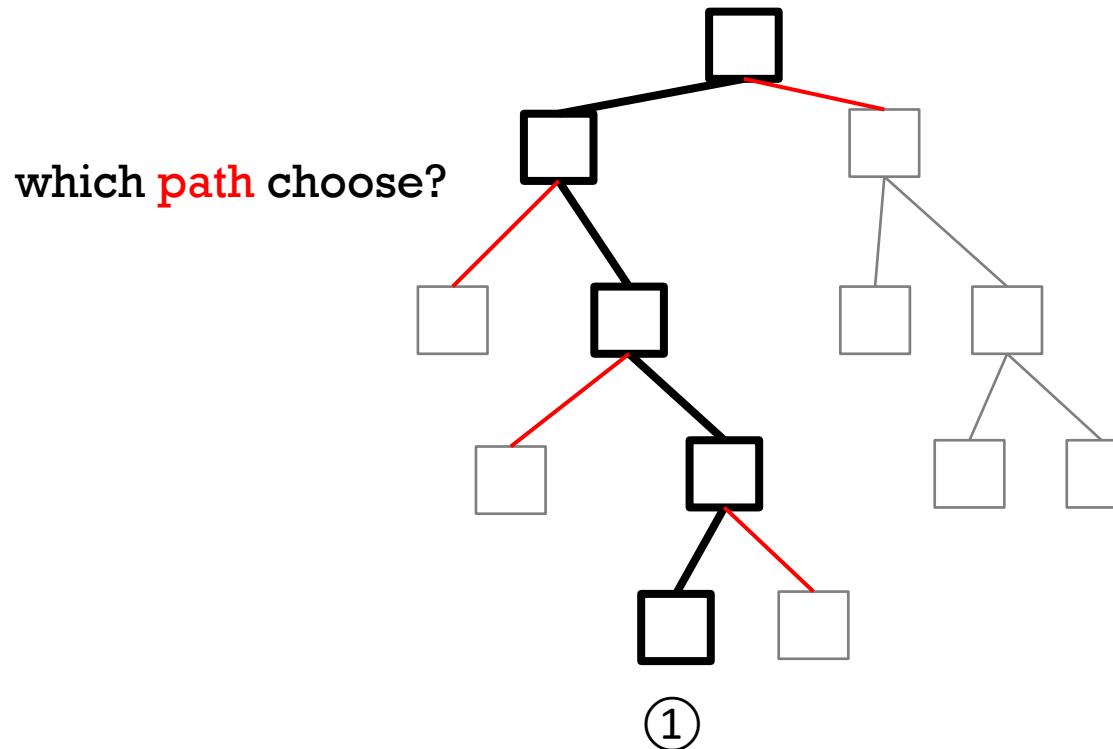
```
void foo(int x, int y) {           void qux(int x) {           void baz(int x) {  
    int a = bar(x);             int a = bar(x);             abs(&x);  
    if (y < 0) ERROR;          if (a > 0) ERROR;  
}                                }                                }  
}                                (a)                                (b)                                (c)
```

- DSEの欠点/限界(通常のStatic Symbolicでも起こり得る)
 - 記号追跡できない関数で伝搬が止まる場合有り(上図a, b, cのbar, abs)
 - システムコール、サードパーティなど
 - 動的だと権限の関係も有り
 - 基本的に計装の機会がなければ追跡できない
 - →false negative (b, c)
 - 実行されていない箇所は当然見逃すことになる
 - →path divergence (c)
 - 制約を解いて生成したインスタンスが期待通りパスを通らないこと
 - ex.) (c)で $x < 0$ を満たすために入力を $x=-2$ にしてもERRORは不発

2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (DSE)

- DSEの注意点
- 再実行で新しい分岐が見つかることがある
 - 未実行分岐の数が大きくなる可能性
→効率的な探索ヒューリスティクスが肝要（2.2節）



2. SYMBOLIC EXECUTION ENGINE

2.1 CONCOLIC EXECUTION (S2E)

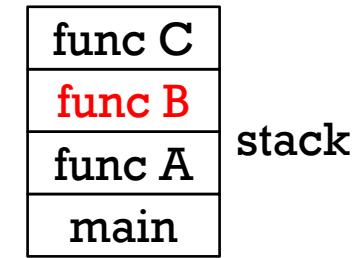
▪ Selective Symbolic Execution (SSE/S2E[29])

- 特定のスタック構成のときだけ（完全）パス探索を行う手法

- Symbolic: 特定のスタック構成
- Concrete: それ以外

- 例: 図のfunc Bがinterestingと仮定

- メソッド: Concrete → Symbolic ($A \rightarrow B$ のように)
 - Bの引数をすべてシンボル化しBを探索
 - 同時にAからの具体的な入力を実行
- メソッド: Symbolic → Concrete ($B \rightarrow C$ のように)
 - Cへの引数を具体化
 - ありえない引数になる可能性 (maybe false positive)
 - Cからの返り値を具体化
 - B内のパス探索が不完全になる可能性(maybe false negative)



2. SYMBOLIC EXECUTION ENGINE

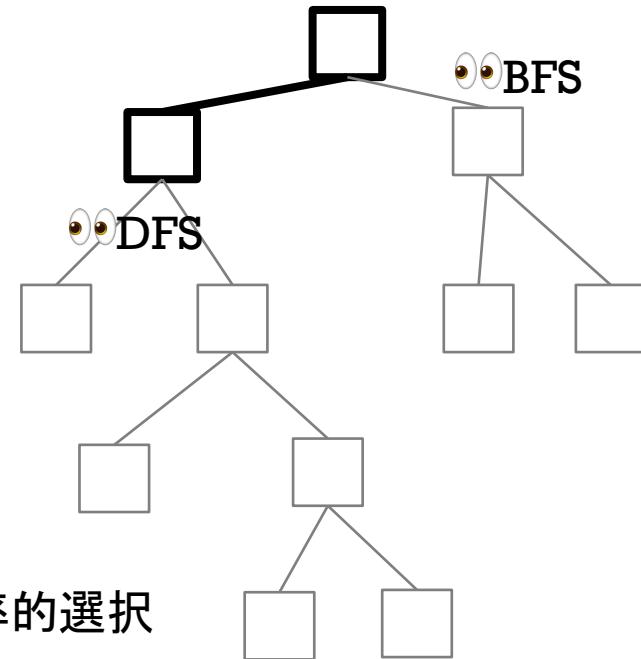
2.2 PATH SELECTION

- パス選択:
 - 有望なパスを見つけるためのヒューリスティクス
 - 普遍的なアルゴリズムはない
 - 何を優先するかで要調整
- Concolicの直後なんで紛らわしいですが、Staticな話も混ざってます

2. SYMBOLIC EXECUTION ENGINE

2.2 PATH SELECTION

- 最も一般的な選択アルゴリズム
 - **Depth-first Search (DFS)**: 深さ優先探索
 - such as DART[51]
 - 利点: メモリ使用量が少ない
 - 欠点: ループ、再帰呼出しには弱い
 - **Breadth-first Search (BFS)**: 幅優先探索
 - 利点: 興味深い動作を素早く見つけやすい
 - 欠点: 深いパスにある脆弱性は見つけにくい
 - **Random path selection**: ランダム選択/確率的選択
 - such as KLEE[20]
 - 各パスを長さと分岐点で評価し、確率的に選択
 - 探索回数が少ないパスを優先し、爆発を抑制
 - 利点、欠点: 用途によってピンキリ



2. SYMBOLIC EXECUTION ENGINE

2.2 PATH SELECTION

- コードカバレッジ最大化 選択アルゴリズム

- such as EXE[21], KLEE[20], Mayhem[25], S2E[29]

- KLEE: 各状態Sに重み付け

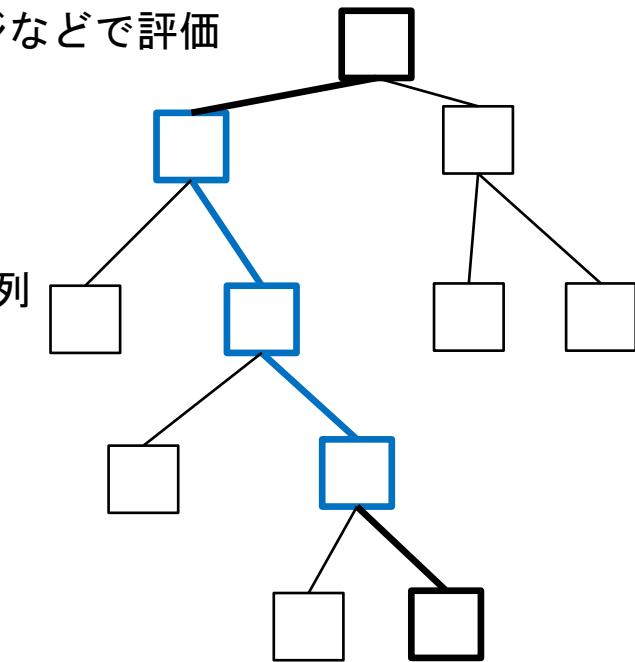
- 重み: 未到達命令までの最短距離、新規カバレッジなどで評価

- Subpath-Guided Search[71]:

- 探索数の少ないサブパスを優先 (in CFG)
 - サブパス: 完全なパスの内、連続した長さ n の部分列
 - 普遍的な n は知られていない

- Shortest-distance[72]:

- ターゲットポイントまでの最短距離で評価



$n = 3$ のサブパス例

[21] EXE: Automatically generating inputs of death [CCS'06]

[25] Unleashing mayhem on binary code [SP'12]

[71] Steering symbolic execution to less traveled paths [OOPSLA'13]

[72] Directed symbolic execution [SAS'11]

2. SYMBOLIC EXECUTION ENGINE

2.2 PATH SELECTION

- その他の選択アルゴリズム
 - AEG[8]:
 - buggy-path-first:
 - 悪用できない(unexploitable)小さなバグを含むパスを選択
 - 潜在的なバグの検出
 - loop exhaustion:
 - ループに到達しているパスを選択
 - ループ内のミスがメモリ関連のバグに繋がるという直感
 - Mayhem[25]:
 - Symbolic addressへの参照があるパスを選択
 - Symbolic instruction pointerが検出されたパスを選択
 - Finite State Machine(FSM)によるガイド[118]:
 - FSMで表現できるプロパティ（メモリ使用、ファイル使用など）を活用
 - プロパティを満たす可能性で判定

[8] AEG: Automatic exploit generation [NDSS'11]

[118] Regular property guided dynamic symbolic execution [ICSE'15]

2. SYMBOLIC EXECUTION ENGINE

2.3 SYMBOLIC BACKWARD EXECUTION

- Symbolic Backward Execution (SBE[26, 40])
 - ⇔ Symbolic Forward Execution
 - 脆弱性を引き起こす箇所からmain entryまでを探索する手法 (!)
- 例: Call Chain Symbolic Backward Execution (CCSBE[72])
 - ① 対象行を持つ関数fのエントリーまでパス探索
 - ② 関数fを呼ぶ関数を一つ選択し、同様にエントリーまでパス探索
 - 以降main entryに到達するまでパス探索と関数選択を繰り返す
- 要件:
 - inter-proceduralなControl-Flow Graphが必要（関数選択用、制約用）
 - ぶっちゃけこれがムズい
- 一般にはスケールしないが、発想自体は面白い
 - (5章でもうちょい触れるらしい)

2. SYMBOLIC EXECUTION ENGINE

2.4 DESIGN PRINCIPLES OF SYMBOLIC EXECUTOR

- 記号実行器の設計原則
 - (1)Progress:
 - 長時間実行してもリソースを食いつぶさないことが望ましい
 - 特にメモリリソース
 - (2)Work repetition:
 - 共通部分の無駄を減らすべし
 - (3)Analysis reuse:
 - 分析済みの結果は可能な限り再利用すべし
 - 特にSMTソルバの実行回数を減らすことが望ましい
 - (1)と(2),(3)は割とトレードオフ
 - どこまで記録して、再利用するのかをよく考える必要性

- 本日はここまで
- ここまで所感
 - パス選択アルゴリズムで気になったモノ（小泉の研究的に）
 - Symbolic Backward Execution [26, 40, 72]
 - Shortest-distance [72]
 - (^ω^)つ[72] Directed symbolic execution [SAS'11]
 - AEG[8]のloop exhaustion
 - ループの判定法が気になる
 - まだループやら関数呼び出しやらは未解消なのでその辺が気になる
 - また今週読みマス

35

前回まで

1. INTRODUCTION 表記について

- $S = (\text{stmt}, \sigma, \pi)$
- stmtによる状態Sの更新
 - 代入 $x = e$:
 - $\sigma \rightarrow \sigma'$, $\text{stmt} \rightarrow \text{stmt} + 1$
 - σ' includes $x \rightarrow e_s$ (本来は|→みたいな矢印)
 - e_s : 実行状態で分けられた代入元の式e、単項or二項
 - 条件分岐 **if e stmt_true else stmt_false** :
 - $\pi \rightarrow \pi'$, $\text{stmt} \rightarrow \text{stmt_next}$
 - $\pi' = \pi \wedge e_s$ ($\text{stmt_next} == \text{stmt_true}$)
 - $\pi' = \pi \wedge \neg e_s$ ($\text{stmt_next} == \text{stmt_false}$)
 - e_s : boolean e を評価して得られる記号式
 - ジャンプ **goto stmt_target**:
 - $\text{stmt} \rightarrow \text{stmt_target}$

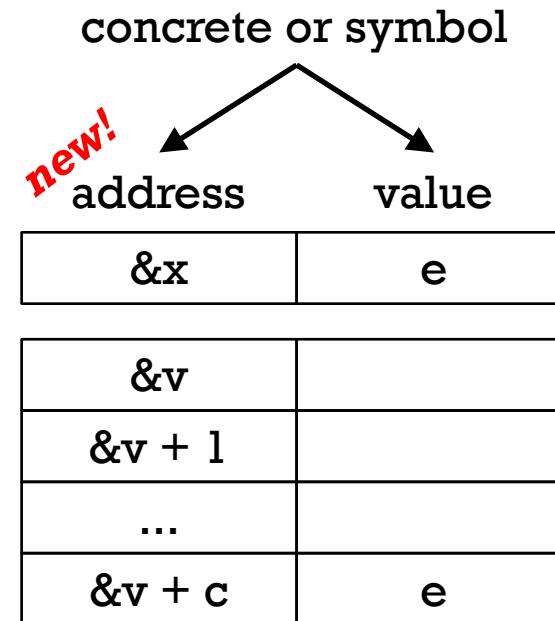
1. INTRODUCTION 例題: SOURCE CODE

- 例題: 8行目のassertでエラーが出る入力組(a, b)の条件を探したい
 - 要するに $x-y == 0$ となるようなa, bは何じゃろな

```
1. void foobar(int a, int b) {  
2.     int x = 1, y = 0;  
3.     if (a != 0) {  
4.         y = 3+x;  
5.         if (b == 0)  
6.             x = 2*(a+b);  
7.     }  
8.     assert(x-y != 0);  
9. }
```

3. MEMORY MODEL

- 前回の例題ではポインタ（アドレス）の概念はなかった
 - 前回の例: foobar
- アドレスもシンボルになりえる
 - →メモリのモデル化 = シンボリックストアσの拡張が必要
- アドレス拡張後のストア表記 (valueのみ)
 - 変数への代入 $x = e$
 - $x \rightarrow e$ ($\&x \rightarrow e$ の意)
 - 配列への代入 $v[c] = e$
 - $v[c] \rightarrow e$ ($\&v + c \rightarrow e$ の意)
- 厄介なのはアドレスのシンボル化
 - ex.) 配列の添字cがシンボル
 - *symbolic memory address* 問題
 - 以降はこれの解決案例の列举



3. MEMORY MODEL

3.1 FULLY SYMBOLIC MEMORY

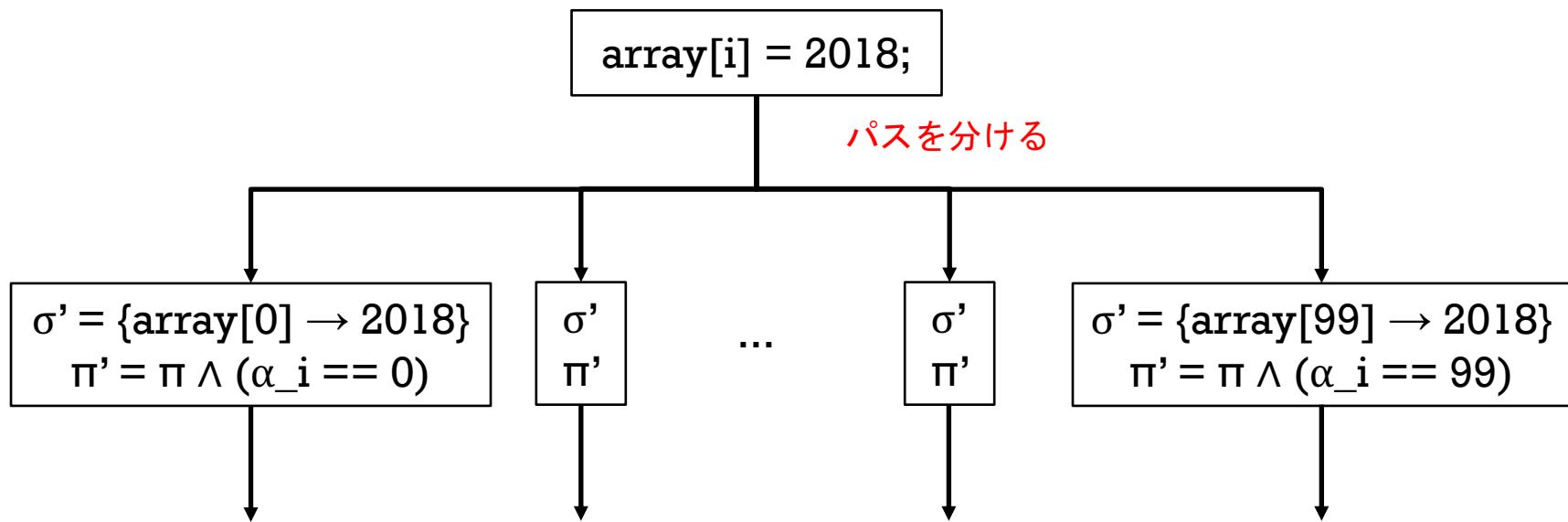
- 最も一般的な解放 → 全メモリアドレスのシンボル化
 - fully symbolic memory
 - 一般的というか単純、愚直なやり方
 - A. State-forking 方式
 - アドレスの全可能性からパスを分岐させる
 - B. if-then-else 方式
 - 制約内に条件分岐を組み込む = ソルバに頼る
- 注: State-forkingはQsym内で説明したforking-basedとは無関係

3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

■ State-forking 方式

- アドレスがとり得る全範囲で場合分け & パスを分ける
- ex.) $\text{array}[i] = 2018;$ で添字 i がシンボル
 $\&\text{array}$ は固定値、 i の範囲が 0~99 の整数値とする
 $\rightarrow \text{array}[0] = 2018; \sim \text{array}[99] = 2018;$ までの可能性でパス分岐



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

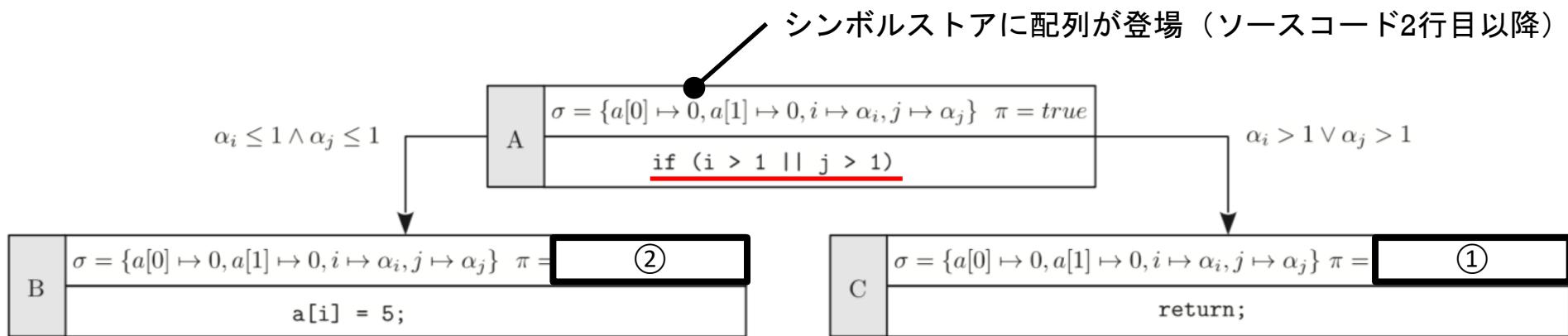
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう

```
1. void foobar(unsigned i, unsigned j) {  
2.     int a[2] = { 0 };  
3.     if (i>1 || j>1) return;  
4.     a[i] = 5;  
5.     assert(a[j] != 5);  
6. }
```

3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

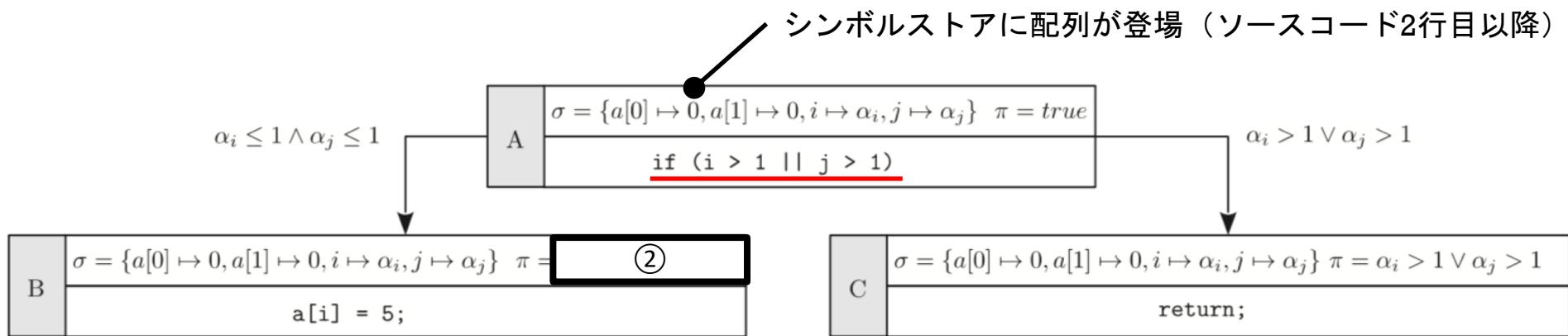
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 1: 3行目の分岐（前回の復習も兼ねて）



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

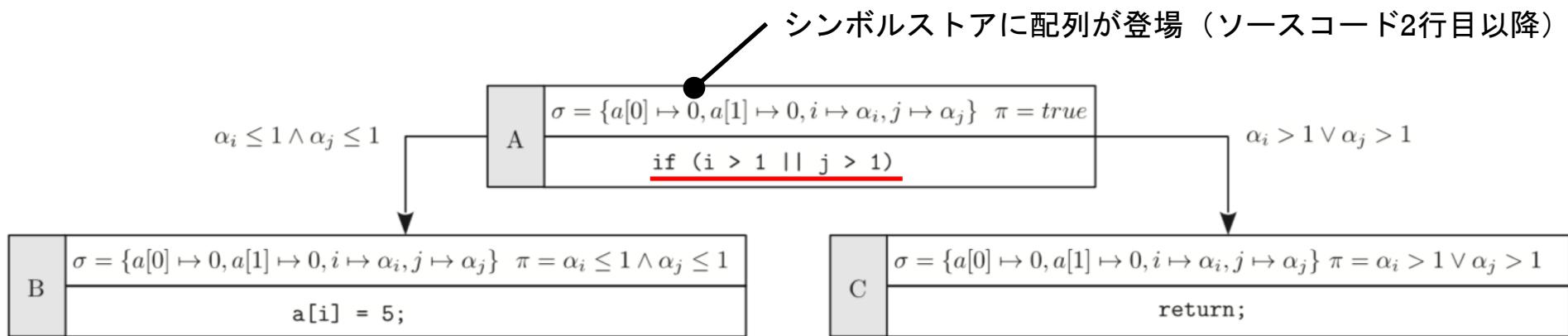
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 1: 3行目の分岐（前回の復習も兼ねて）



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

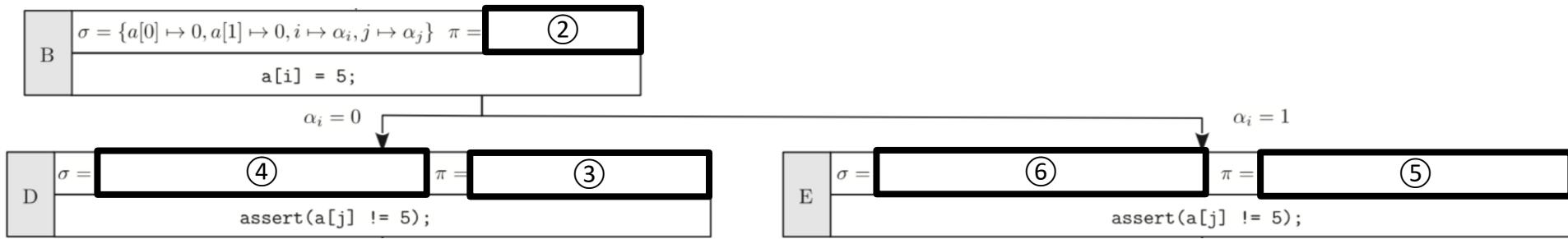
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 1: 3行目の分岐（前回の復習も兼ねて）



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

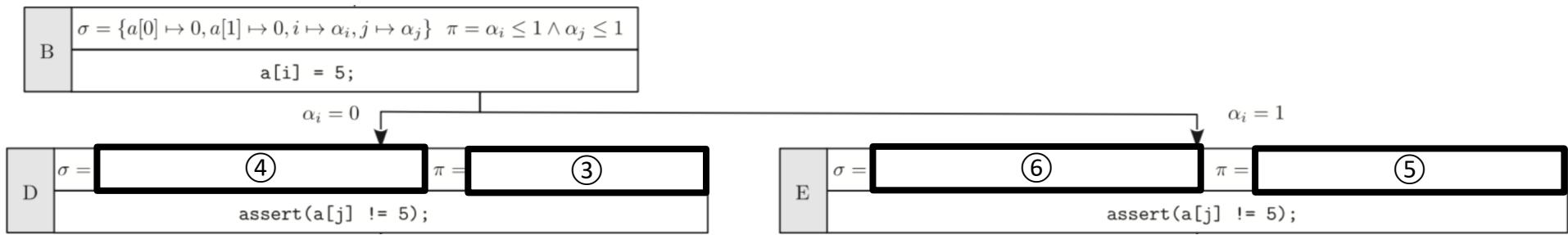
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

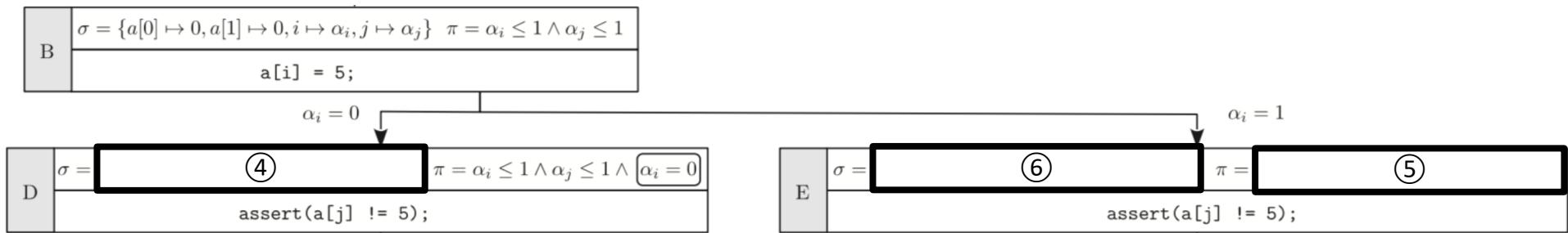
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

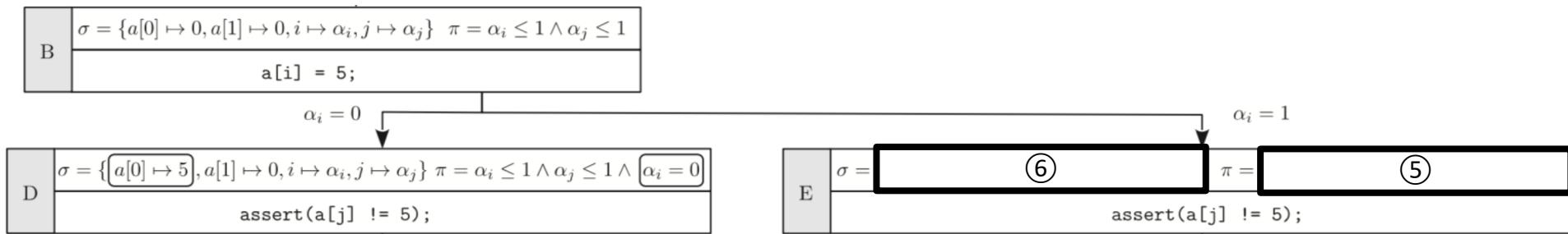
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

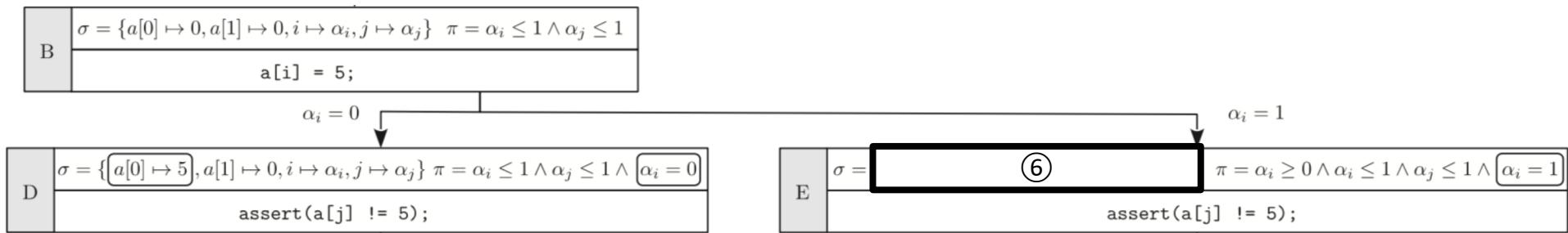
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

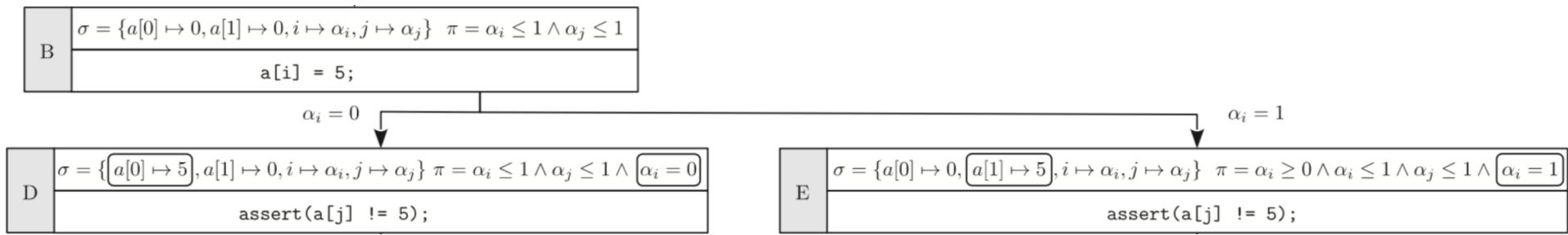
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

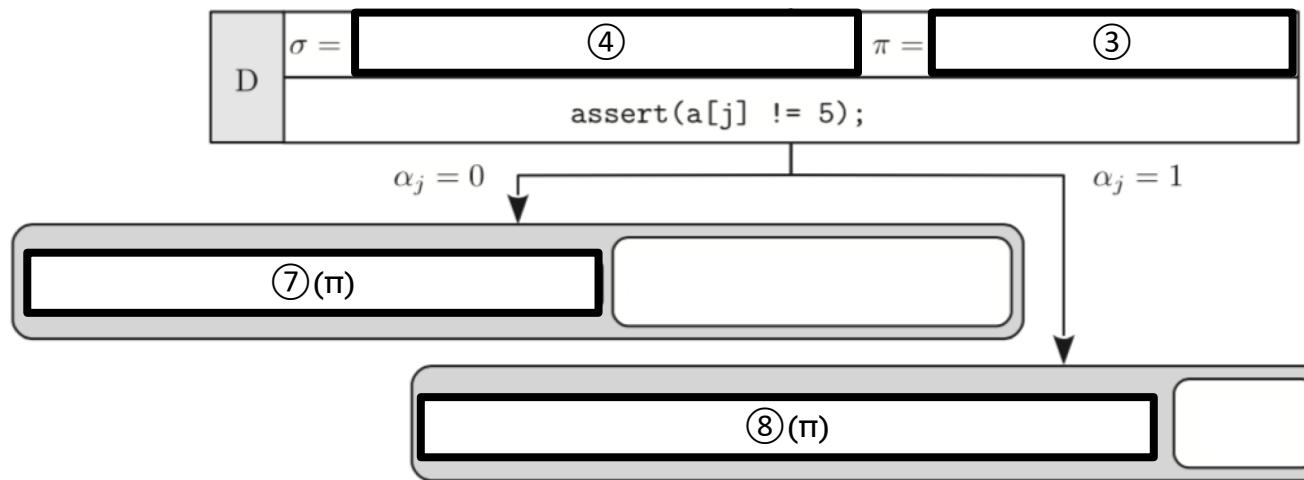
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 2: 4行目 配列への代入



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

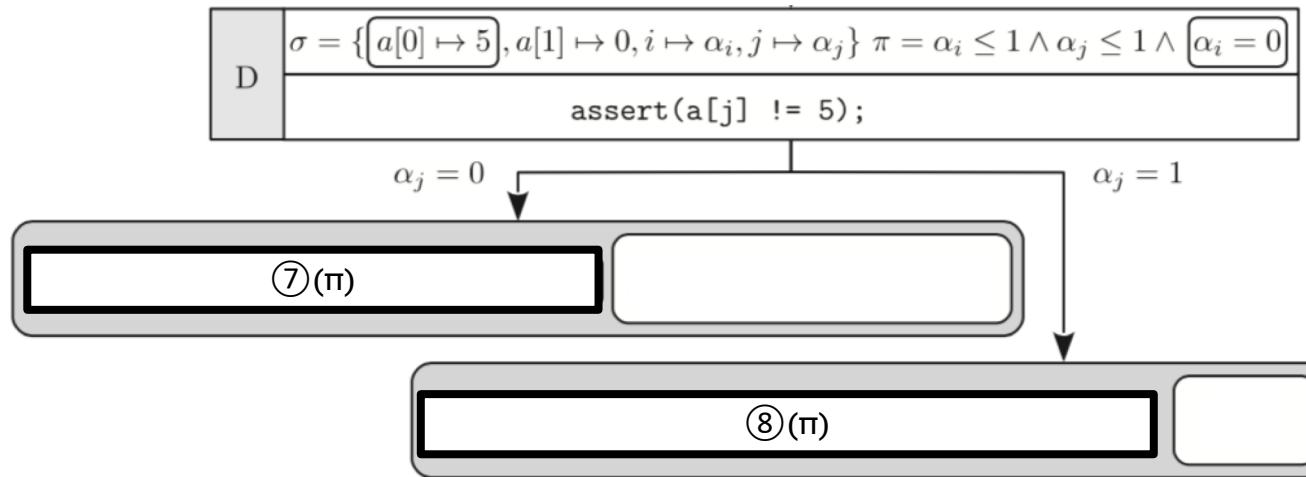
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 3: 5行目のassert (i = 0の場合)



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

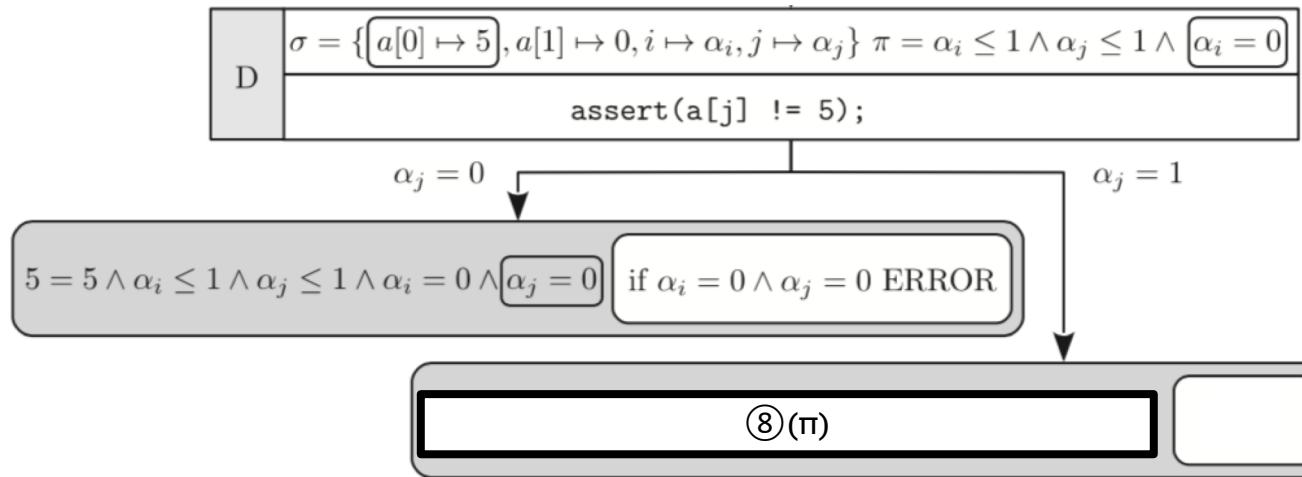
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 3: 5行目のassert (i = 0の場合)



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

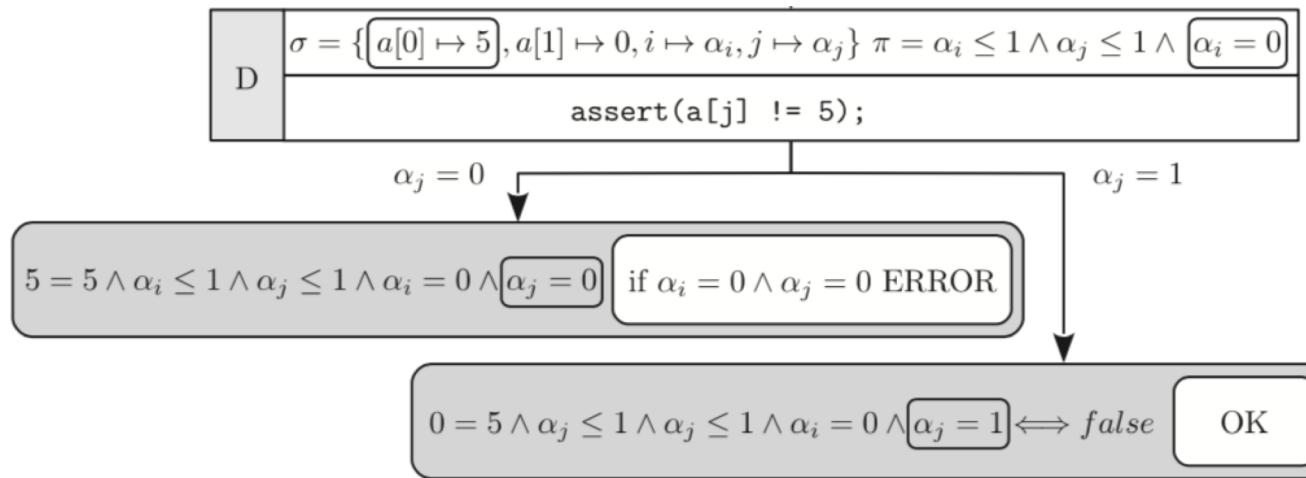
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 3: 5行目のassert (i = 0の場合)



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

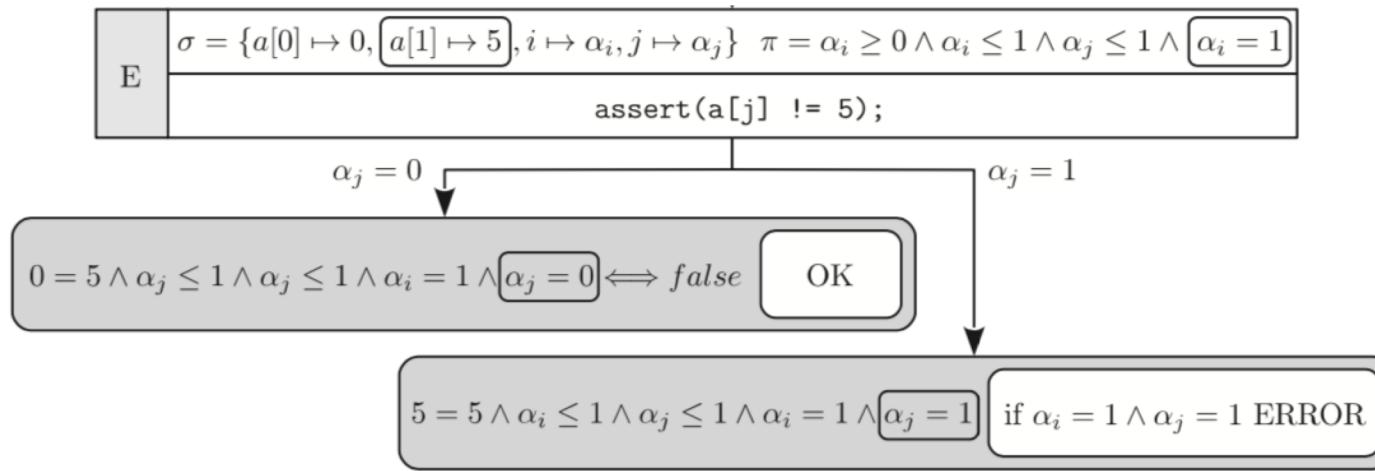
- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 3: 5行目のassert (i = 0の場合)



3. MEMORY MODEL

3.1 FULLY (STATE-FORKING)

- 例題: assertでエラーが出る入力の組(i, j)を見つけよう
 - step 3': 5行目のassert (i = 1の場合)
 - 同様にjの値で場合分け



3. MEMORY MODEL

3.1 FULLY (IF-THEN-ELSE)

■ if-then-else方式

- 制約式に $\text{ite}(c, t, f)$ を組み込む (ソルバがサポートしてなければムリ♪)

- $\text{ite}(c, t, f)$: c が真なら t の値、それ以外なら f の値をとる式
 - 要は三項演算子

■ メモリ読み込み時

- メモリの値: $\text{ite}(\alpha == a_1, \sigma(a_1), \text{ite}(\alpha == a_2, \sigma(a_2), \text{ite}(\dots)))$
 - α : シンボルアドレス
 - a_k : とり得る具体値
 - $\sigma(a_k)$: ストアで対象アドレスに格納されている値 ($a[0] \rightarrow 5$ の5とか)

■ メモリ書き込み時

- 書き込み得る各アドレスについて
 - $\sigma(a_k) \rightarrow \text{ite}(\alpha == a_k, e, \sigma(a_k))$
 - 意訳: 書き込み対象になった場合とならなかった場合の三項演算子をシンボリックストアに格納

3. MEMORY MODEL

3.1 FULLY (IF-THEN-ELSE)

■ if-then-else方式

- 制約式に $\text{ite}(c, t, f)$ を組み込む (ソルバがサポートしてなければムリ♪)

- $\text{ite}(c, t, f)$: c が真なら t の値、それ以外なら f の値をとる式
 - 要は三項演算子

■ メモリ読み込み時

- ex.) $\text{array}[i] > 0$ という条件式の評価
 i がシンボル、 $\&\text{array}$ は固定値、 $i = \{0, 1\}$ とする
 $\text{array}[0] := 10 \quad \text{array}[1] := -5$ とする (適当な初期値)
 $\rightarrow \text{array}[i]: \text{ite}(\alpha_i == 0, \sigma(\text{value}: 0), \text{ite}(\alpha_i == 1, \sigma(\text{value}: 1), ()))$
 $= \text{ite}(\alpha_i == 0, \text{array}[0], \text{array}[1]) > 0$
 $= \underline{\text{ite}(\alpha_i == 0, 10, -5)} > 0$ という制約が追加

■ メモリ書き込み時

- ex.) 上と同じ前提で $\text{array}[i] = 25;$ という代入 (書き込み) を評価
 $\rightarrow \sigma' = \{ \text{array}[0] \rightarrow \text{ite}(\alpha_i == 0, 25, \text{array}[0]),$
 $\text{array}[1] \rightarrow \text{ite}(\alpha_i == 1, 25, \text{array}[1]) \}$

3. MEMORY MODEL

3.1 FULLY (IF-THEN-ELSE)

- 例題アゲイン: assertでエラーが出る入力の組(i, j)を見つけよう

```
1. void foobar(unsigned i, unsigned j) {  
2.     int a[2] = { 0 };  
3.     if (i>1 || j>1) return;  
4.     a[i] = 5;  
5.     assert(a[j] != 5);  
6. }
```

3. MEMORY MODEL

3.1 FULLY (IF-THEN-ELSE)

- 例題アゲイン: assertでエラーが出る入力の組(i, j)を見つけよう

- step 1: B→D メモリ書き込み

- $a[0] \rightarrow 0 \rightarrow a[0] \rightarrow \text{ite}(\alpha_i == 0, 5, 0)$ 添字iが0なら5, 違うなら0 の意
 - $a[1] \rightarrow 1 \rightarrow a[1] \rightarrow \text{ite}(\alpha_i == 1, 5, 0)$

- step 2: D→assert メモリ読み込み

- $a[j] := \text{ite}(\alpha_j == 0, a[0], a[1])$

B	$\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
	$a[i] = 5;$



D	$\sigma = \{a[0] \mapsto \text{ite}(\alpha_i = 0, 5, 0), a[1] \mapsto \text{ite}(\alpha_i = 1, 5, 0)\}, i \mapsto \alpha_i, j \mapsto \alpha_j$ $\pi = \alpha_i \leq 1 \wedge \alpha_j \leq 1$
	$\text{assert}(a[j] \neq 5);$



$a[j]$	$\text{ite}(\alpha_j = 0, \text{ite}(\alpha_i = 0, 5, 0), \text{ite}(\alpha_i = 1, 5, 0)) = 5 \wedge \alpha_i \leq 1 \wedge \alpha_j \leq 1$	$\text{if } (\alpha_i = 0 \wedge \alpha_j = 0) \vee (\alpha_i = 1 \wedge \alpha_j = 1) \text{ ERROR}$
--------	--	---

3. MEMORY MODEL

3.1 FULLY SYMBOLIC MEMORY

- 最も一般的な解放 → 全メモリアドレスのシンボル化
 - fully symbolic memory
- 今示した例ではアドレスの範囲は非常に狭い
 - 実際には任意の（アドレス空間全体の）アドレスをとり得る
 - 例に出した`&array`のアドレスも本来不定値
 - `&array[i]`のアドレスはアドレス空間全体に分布し得る
 - →爆発（パス爆発 or SMTソルバが短時間で解けない）
 - 以降の節（3.2節～3.4節）：スケールするためのテクニックの紹介
 - 各テクニックは以下のいずれかを考慮
 - ①メモリ表現のコンパクト化
 - ②健全性とパフォーマンスのトレードオフ
 - ③ヒープモデリング

3. MEMORY MODEL

3.2 ADDRESS CONCRETIZATION

- アドレスの具体化
 - 利点: 爆発の抑制、制約の簡易化
 - 欠点: アドレス値に依存するパスを見逃す可能性
 - それこそ配列関連とか
- DART[51], CUTE[91]
 - T*型の参照を具体化、型サイズによる幅有り
 - DART: アドレスはランダムに決定
 - CUTE: 初回実行ではNull、以降は具体値
- 注意点:
 - 型が構造体なら、各フィールドへ具体値が割り当てられる
 - 実行間でアドレスが異なる可能性
→記号式には論理アドレスを使用

[51] DART: Directed automated random testing [PLDI'05]

[91] CUTE: A concolic unit testing engine for C [ECSE/FSE'13] (2005)

3. MEMORY MODEL

3.3 PARTIAL MEMORY MODELING

- ハイブリッド方式 (Mayhem[25], Angr[95])
 - fully symbolic memory (3.1)
address concretization (3.2) の混成

- モチベ:
 - 全シンボル化はスケールしない
 - アドレスの具体化は健全性を損なう（パスを見落とす）可能性
 - →Mayhem[25]: 両者のバランスを思案

3. MEMORY MODEL

3.3 PARTIAL MEMORY MODELING

- 例: Mayhem方式
 - 書き込み先アドレスは常に具体化
 - 読み込み先アドレスは一部記号化
 - 読み込み可能アドレスの間隔が小さい場合に限る
- 基本的なアドレス値の範囲の決定法
 - 厳密な範囲が見つかるまで異なる具体値を繰り返し試す & 制約を満たすかチェック
 - →制約を満たす範囲をスキャンするのデス...
- 注意点:
 - シンボリックアドレスへのアクセスごとにソルバを呼ぶのは高価
 - メモリの範囲が非連続な可能性もある
 - 構造体が絡むと厄介
→VSA[42]やクエリのキャッシングで最適化

3. MEMORY MODEL

3.4 LAZY INITIALIZATION

- 状態Sの拡張: ヒープの情報を追加
 - list, treeなどのデータ構造も扱えるようにする
 - JavaやC++などのOOP言語を想定
- **Lazy Initialization** (遅延初期化 [66],[107]):
 - ヒープオブジェクト生成時ではなくアクセス時に初期化を行う
 - ① Null
 - ② new object
 - ③ exist object
 - 3パターンを考慮して分岐
- 初期化を遅らせることで、必要ない分岐を減らせる可能性
 - 次に例を提示

3. MEMORY MODEL

3.4 LAZY INITIALIZATION

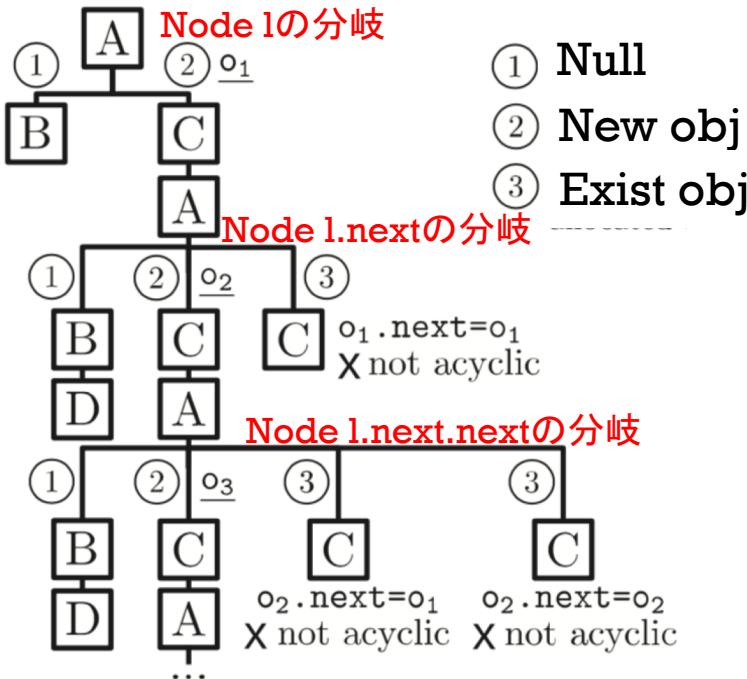
- 例: linked-listへの追加メソッド (再帰型)
 - Aブロックで最初に参照されるのでそこで初期化 (パターン分け)
 - 最初のパターン分けでは③は考慮の必要なし (既存objがないため)
 - l.next以降でも③は事前情報から考慮の必要なし
 - linked-listなら循環=既存objを指すことはないため

```

class Node {
    protected int value;
    protected Node next;

    // precondition: "acyclic structure"
    public Node add(Node l, int v) {
        A|   if (l == null) {
        B|       l = new Node();
        C|       l.value = v; return l;
        D|
    }
}

}
  
```



無限に再帰するので上限を設ける必要有り

4. INTERACTION WITH THE ENVIRONMENT

- ほとんどのプログラムは自己完結しない
 - → エンジンは様々な(外部)環境要素を考慮しなければならない
 - ex.) ファイルシステム、環境変数、ネットワークなどのOS機能
= システム環境
 - ex.) SwingやAndroidなどのアプリケーションフレームワーク
= アプリケーション環境

4. INTERACTION WITH THE ENVIRONMENT

SYSTEM ENVIRONMENT

- 外部環境
 - 何もしなければ、記号追跡できない (2.1節)
 - 追跡できない場合、パス制約が不完全になる可能性がある
- 紹介される対策法
 - Black-box化
 - モデル化
 - その他（仮想化）

4. INTERACTION WITH THE ENVIRONMENT SYSTEM ENVIRONMENT (BLACK BOX)

- 初期の研究 (EXE[21], DART[51], CUTE[91])
 - Black-box化:
 - システムコールなどに具体的な値を使用、内部の制約を見ない
 - = 実際に実行する
 - 利点: 現実的な時間にスケールする
 - 欠点①: 観測可能な挙動に制限がある
 - システムコール内を探索しないため
 - 欠点②: 副作用がある場合、パス探索が不完全になる可能性がある
 - 特にオンライン実行器では注意
 - オンライン: 1回の実行で複数のパスを探索する方式
 - ≈ forking-basedみたいなイメージ

4. INTERACTION WITH THE ENVIRONMENT SYSTEM ENVIRONMENT (MODELING)

- Black-box化の欠点の克服
 - →モデル化 (KLEE[20], AEG[8], CLOUD9[18])
- KLEE: symbolic file system
 - ファイルのモデル化
 - symbolic files: ユーザ定義のn個の記号的ファイル
入力ファイルは操作によってn+1の状態に変化する
(symbolic files or 予期せぬエラー の計n+1状態)
 - システムコールのモデル化
 - ライブラリ粒度のモデル化は時間がかかるので断念
- AEG: KLEE+ α
 - ファイルシステム、ネットワークソケット、環境変数などをモデル化
 - 70以上のライブラリ、システムコールをモデル化
- CLOUD9: AEG+ α (?)
 - POSIXライブラリもモデル化

[20] KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs [OSDI'08]

[8] AEG: Automatic exploit generation [NDSS'11]

[18] Parallel symbolic execution for automated real-world software testing [EuroSys'11]

4. INTERACTION WITH THE ENVIRONMENT SYSTEM ENVIRONMENT (OTHER)

- モデル化への異議あり
 - S2E[29]: モデルの記述は高価なわりに精度に貢献するのは稀では?
(保守作業も必要だでよ)
 - →QEMU, x86-to-LLVM lifterを用いて仮想化
 - 独立したパス間で影響が伝播しないように

4. INTERACTION WITH THE ENVIRONMENT

APPLICATION ENVIRONMENT

- アプリケーション環境
 - VMと言ってもいいかも
 - VM依存の動作は外部環境とみなせる
JNIとかリフレクションとか
- 対策案① Black-box化（引数の具体化）
 - システム環境と同様に、探索が不完全化する可能性
- 対策案② モデル化
 - システム環境よりも厳しい印象
 - 内部が複雑な可能性
 - 探索とは無関係な要素が存在 (call back トリガ用のボタン表示メソッドなど)
 - ソースコードが手に入らなければ簡易モデルを作るのも困難
 - 対closure-source components
 - プログラムスライシングで抽象モデルの構築[24, 105]
 - プログラム合成でデザインパターンのインスタンス化[63]

[24] Automated generation of model classes for java pathfinder [SIGSOFT SEN'14]

[105] Generation of library models for verification of android applications [SIGSOFT SEN'15]

[63] Synthesizing framework models for symbolic execution [ICSE'16]

72

前回まで

1. INTRODUCTION 表記について

- $S = (\text{stmt}, \sigma, \Pi)$
- stmtによる状態Sの更新
 - 代入 $x = e$:
 - $\sigma \rightarrow \sigma'$, $\text{stmt} \rightarrow \text{stmt} + 1$
 - σ' includes $x \rightarrow e_s$ (本来は|→みたいな矢印)
 - e_s : 実行状態で分けられた代入元の式e、単項or二項
 - 条件分岐 **if e stmt_true else stmt_false** :
 - $\Pi \rightarrow \Pi'$, $\text{stmt} \rightarrow \text{stmt_next}$
 - $\Pi' = \Pi \wedge e_s$ ($\text{stmt_next} == \text{stmt_true}$)
 - $\Pi' = \Pi \wedge \neg e_s$ ($\text{stmt_next} == \text{stmt_false}$)
 - e_s : boolean e を評価して得られる記号式
 - ジャンプ **goto stmt_target**:
 - $\text{stmt} \rightarrow \text{stmt_target}$

3. MEMORY MODEL

3.1 FULLY (IF-THEN-ELSE)

■ if-then-else方式

- 制約式に $\text{ite}(c, t, f)$ を組み込む (ソルバがサポートしてなければムリ♪)

- $\text{ite}(c, t, f)$: c が真なら t の値、それ以外なら f の値をとる式
 - 要は三項演算子

■ メモリ読み込み時

- メモリの値: $\text{ite}(\alpha == a_1, \sigma(a_1), \text{ite}(\alpha == a_2, \sigma(a_2), \text{ite}(\dots)))$
 - α : シンボルアドレス
 - a_k : とり得る具体値
 - $\sigma(a_k)$: ストアで対象アドレスに格納されている値 ($a[0] \rightarrow 5$ の5とか)

■ メモリ書き込み時

- 書き込み得る各アドレスについて
 - $\sigma(a_k) \rightarrow \text{ite}(\alpha == a_k, e, \sigma(a_k))$
 - 意訳: 書き込み対象になった場合とならなかった場合の三項演算子をシンボリックストアに格納

5. PATH EXPLOSION

- パス爆発の主要因: ループと関数呼び出し
 - ループでの爆発例:

```
int x = sym_input();
while(x > 0) x = sym_input();
```

- 1章でのstmt定義に分解するなら、ループはif-gotoが延々と続く
- ループした回数をkとするなら、制約 π は
 - $\pi_k = (\wedge_{i \in [1,k]} \alpha_{x_i} > 0) \wedge (\alpha_{k+1} \leq 0)$ となる
 - $\rightarrow k$ 通りの制約とそれに対応するパスが存在 (k の上限がなければ無限)
- 一般的な対ループ策: ループの上限を設定する

5. PATH EXPLOSION

- パス爆発の主要因: ループと関数呼び出し
 - 以降の節では対パス爆発のアプローチを紹介
 - 5.1: unsatなパスを間引く
 - 5.2: 関数、ループのサマリを作る
 - 5.3: パスの包含関係、等価性を見つける
 - 5.4: Under-constrained Symbolic Execution
 - 5.5: Preconditionや入力特徴の利用
 - 5.6: 状態マージ
 - 5.7: 各種プログラム解析、最適化技術の利用

5. PATH EXPLOSION

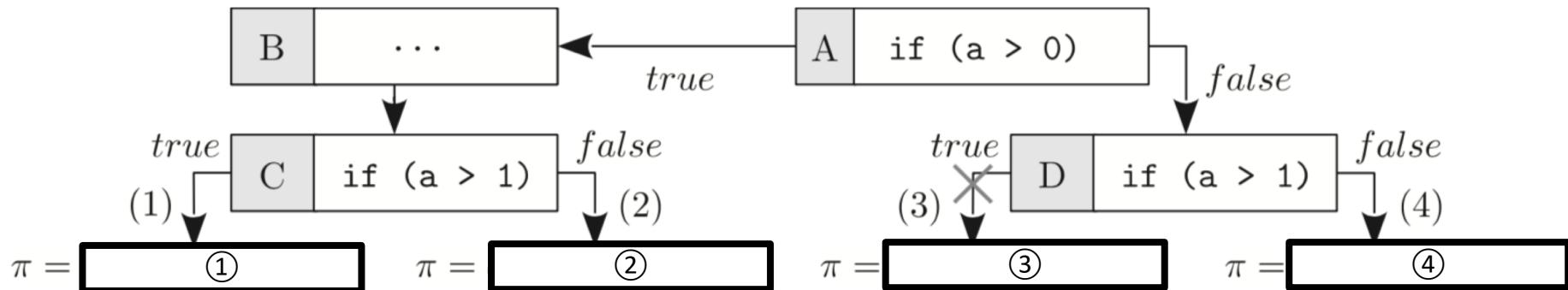
5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない

```
if (a > 0) { ... }
```

```
if (a > 1) { ... }
```



5. PATH EXPLOSION

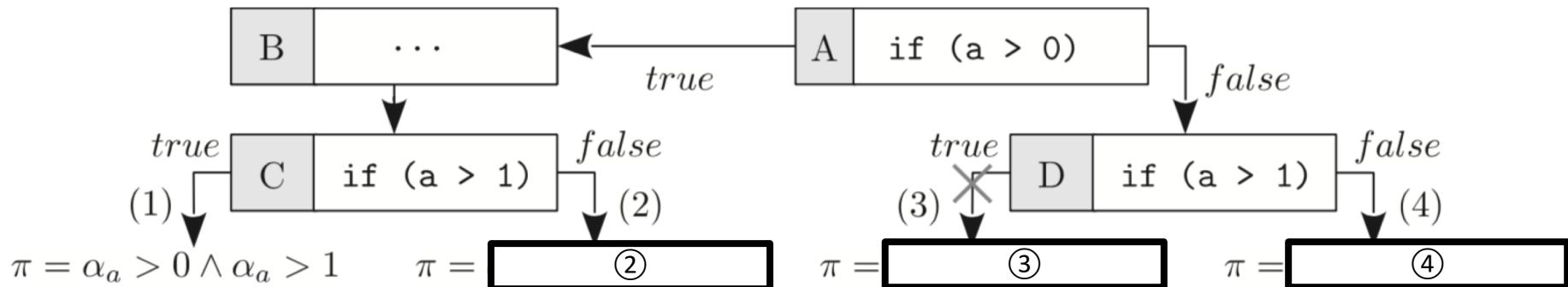
5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない

```
if (a > 0) { ... }
```

```
if (a > 1) { ... }
```



5. PATH EXPLOSION

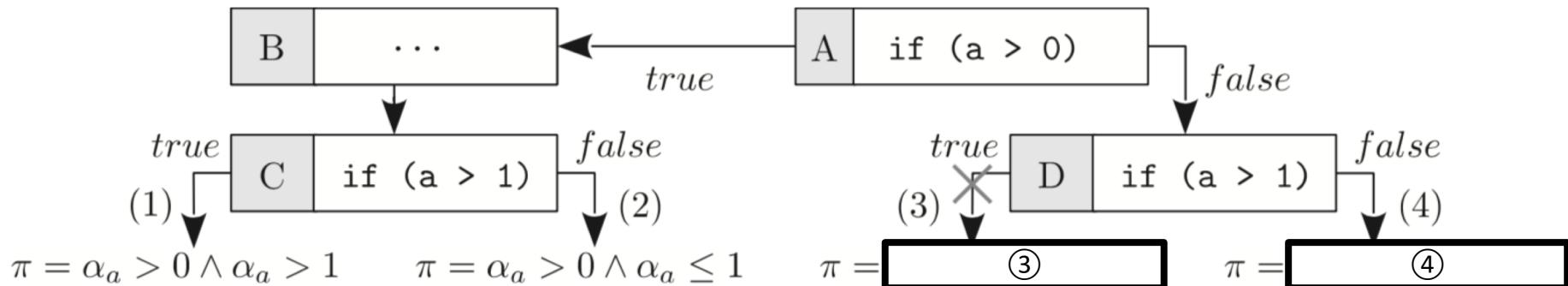
5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない

```
if (a > 0) { ... }
```

```
if (a > 1) { ... }
```



5. PATH EXPLOSION

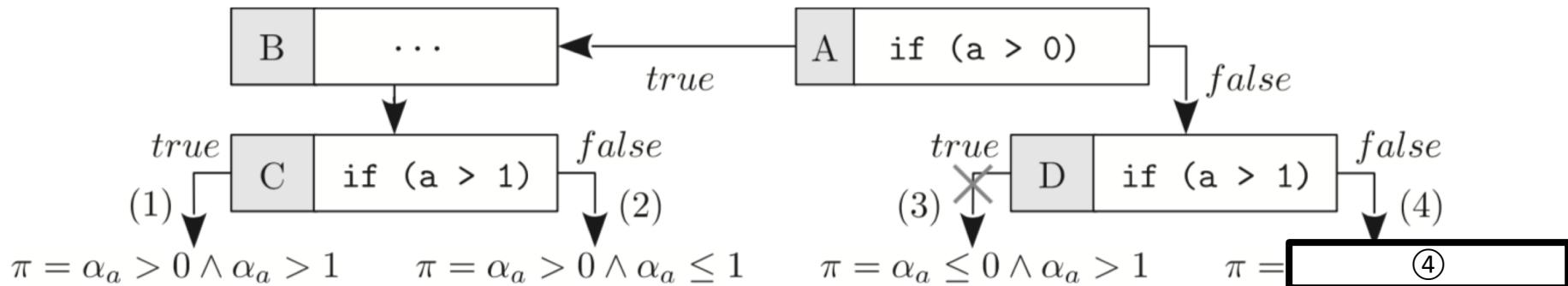
5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない

```
if (a > 0) { ... }
```

```
if (a > 1) { ... }
```



5. PATH EXPLOSION

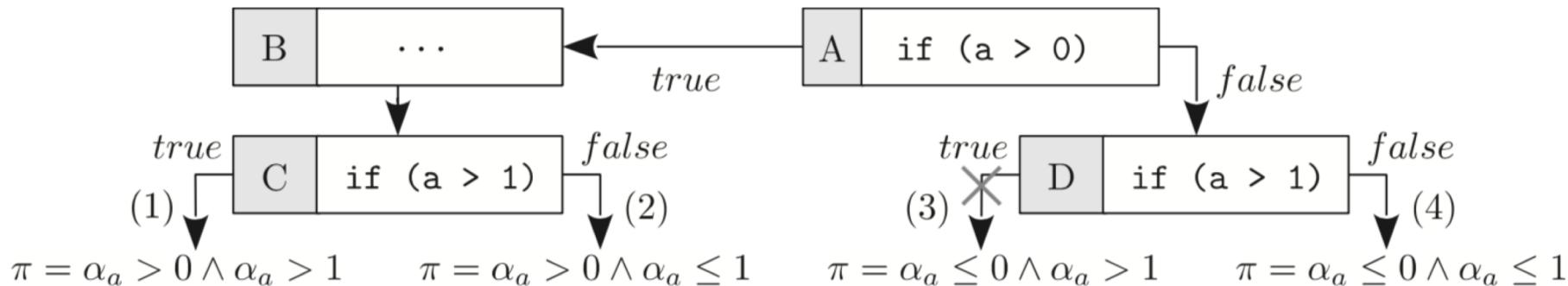
5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない

```
if (a > 0) { ... }
```

```
if (a > 1) { ... }
```

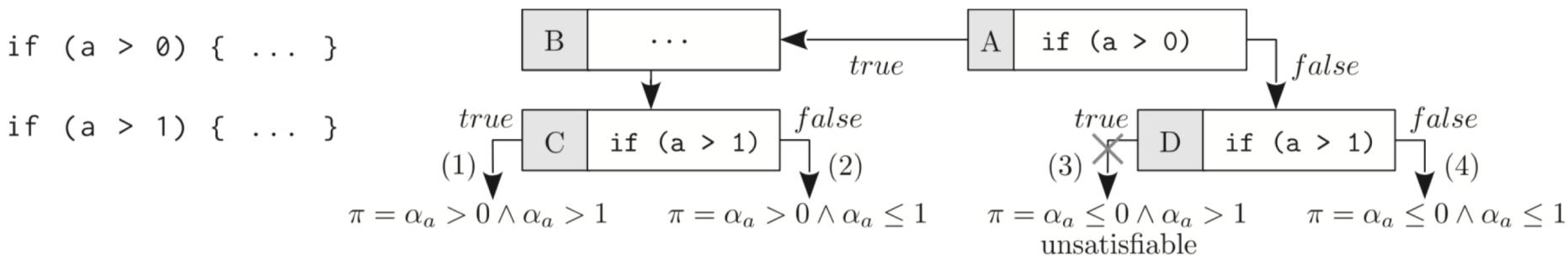


5. PATH EXPLOSION

5.1 PRUNING UNREALIZABLE PATHS

- 実現不可能なパスの間引き

- SMTソルバに制約を解かせる
→unsatisfiableな制約を持つパスは以降の探索を行わない



- 例: $a > 0$ を満たさないパスでは $a > 1$ がtrueにはなりえない(unsat)

- 評価のタイミングで2手法

- eager evaluation : 分岐の都度SMTソルバを呼ぶ
- lazy evaluation : 特定のタイミングでソルバを呼ぶ
(検体でエラーが発生したときなど)

- QSYMは一部のunsatなパスを利用 (Optimistic Solving)

5. PATH EXPLOSION

5.2 FUNCTION AND LOOP SUMMARIZATION

- code fragment (関数 or ループ本体)が複数回実行
 - →再利用のために要約(summary)を生成
- 関数 f の要約(function summary)[50]
 - (for Concolic Execution)
 - パス w の探索中に以下の2点をキャプチャ
 - 関数 f の引数、出力に対する制約 φ_w
 - 関数 f の副作用
 - 関数のサマリ $\rightarrow \text{Summary}(f) = \bigvee \varphi_w$
 - 右の例 (from [50])
 - 期待されるサマリ
$$\text{Summary}(\text{is_positive}) = (x > 0 \wedge \text{ret} = 1) \vee (x \leq 0 \wedge \text{ret} = 0)$$

```
int is_positive(int x){  
    if(x > 0) return 1;  
    return 0;  
}
```

5. PATH EXPLOSION

5.2 FUNCTION AND LOOP SUMMARIZATION

■ ループの要約

- ループ条件と記号変数間の依存関係を推論
 - → pre-conditionとpost-conditionを使用
- 初期の研究: 繰り返し全体で記号変数を更新するループのみに対応
 - Xネストループ
 - Xマルチパスループ(内側に分岐があるループ)
- Proteus[113]:
 - マルチパスループに対応
 - パス条件とループ内パスの規則性の有無で分類
 - ループ条件が帰納的か (IV条件) 、非帰納的か (NIV条件)
 - 特定のパスの実行が逐次的か、規則的か、不規則的か

```
int n:=*;
int x:=*;
int z:=*;
while(x<n)
  if(z>x) x++;
  else z++;
assert(x==z);
```

```
while(i<100){
  if(a<=5) a++;
  else a-=4;
  if(j<8) j++;
  else j-=3;
  i++;
}
```

```
while(i<LINT){
  int j=nondet();
  assume(1<=j);
  assume(j<LINT);
  i = i + j;
  k++;
}
```

```
while(x1>0&&x2>0&&x3>0)
  if(c1) x1=x1-1;
  else if(c2) x2=x2-1;
  else x3=x3-1;
  c1=nondet(); c2=nondet();
  assert(x1==0||x2==0
         ||x3==0);
```

```
while(i<A&&j<B)
  if(A[i]==B[j])
    i++;
    j++;
  else
    i=i-j+1;
    j=0;
```

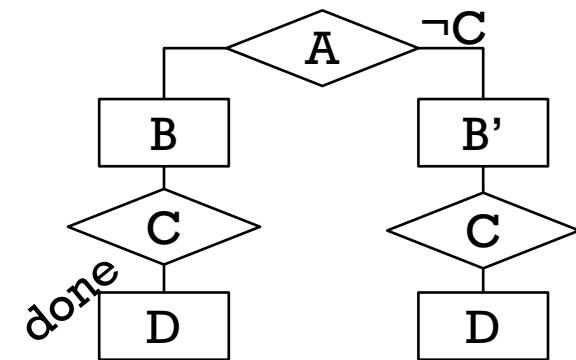
```
int s=1,x1=x2=0;
while(nondet())
  if(s==1) x1++;
  else if(s==2) x2++;
  s++;
  if(s==5) s=1;
  if(s==1&&x1!=x2) ERROR;
```

5. PATH EXPLOSION

5.3 PATH SUBSUMPTION AND EQUIVALENCE

- 補間(interpolation):

- アプリケーション内のパスの冗長性
 - →既に同様のパスを検査済みならば、似たパスの検査を省けるのでは？
 - →期待する動作へ到達しない条件を推論
条件を満たしたパスの探索を打ち切る



- 補間による包含関係の推論

- annotation algorithm[75]:
 - 分岐とステートメントに対してラベル付
特定のラベルへ到達するパスは目的地に到達しないと判断
- post-conditioned symbolic execution[117]:
 - 完全探索されたパスから、逆方向にも制約を伝播
 - 各分岐でpost-conditionの否定を制約に追加
 - パスが以前の探索に包含していたら、制約がunsatになる

[75] Lazy annotation for program testing and verification [CAV'10]

[117] Postconditioned symbolic execution [ICST'15]

5. PATH EXPLOSION

5.3 PATH SUBSUMPTION AND EQUIVALENCE

- 無限ループ
 - 無限ループが存在 → 無数のパスが無限ループを介して発生
 - → 健全性の悪化
 - 紹介されている関連研究は基本的にループ不变量の計算を主眼
 - such as [75, 61, 60, 62]
- 抽象化による包含関係[6]
 - スカラ型だけでなく、データ構造への操作も状態Sに加える
 - 状態数は（実質的に）無限
 - →有限になるように抽象化

[61] Unbounded symbolic execution for program verification [RV'11]

[60] TRACER: A symbolic execution tool for verification [CAV'12]

[62] An interpolation method for CLP traversal [CP'09]

[6] Symbolic execution with abstraction [STTT'09]

5. PATH EXPLOSION

5.3 PATH SUBSUMPTION AND EQUIVALENCE

- パス分割 (Path Partitioning)
 - 制御フロー解析、データフロー解析から新規性のないパスのフィルタリングように粗い関係性を求める
- [74]:
 - 互いに干渉しないブロックでのConcolic実行の入力を分ける
- [84]:
 - 出力に焦点
 - 出力に対して同一のスライスを持つ → 2つのパスは同一のパーティション
- [109]:
 - 記号入力による導出をキャプチャ
 - →個々のステートメントのスライスを作製
 - →依存関係解析でスライスの等価性を効率的にチェック

[74] Reducing test inputs using information partitions [CAV'09]

[84] Path exploration based on symbolic output [TSEM'13]

[109] Dependence guided symbolic execution [IEEE/TSE'17]

5. PATH EXPLOSION

5.4 UNDER-CONSTRAINED SYMBOLIC EXECUTION

- プログラムを細かく分けて、個々に調べる
 - 関数単位で分ければ、intra-proceduralな解析をする、と言えるかも
 - ただし、個々に調べただけではpreconditionといった情報が欠落
 - 例えば関数の引数にありえない値が入ったり
- Under-Constrained Symbolic Execution[45]:
 - 分割したプログラムブロック内の記号変数が無制約
 - →under-constrainedとしてマーク
 - →under-constrainedな変数についての制約を収集してpreconditionなどを復元？（すいません理解できません）
 - Surveyの評価:
 - エラーを見逃す可能性があるが、大規模なプログラムでは興味深い結果を生む
 - 小泉の直感:
 - 厄介な箇所(一部の関数とかループとか)をスキップするために使用するらしい
→ループを踏み抜きたい小泉的には相反するアイデアかも

5. PATH EXPLOSION

5.5 EXPLOITING PRECONDITIONS AND INPUT FEATURES

- 入力プロパティの活用

- Preconditioned symbolic execution[8]
 - 事前に決定している条件を制約に組み込む
 - 具体的にはπの初期値をtrueからpreconditionにする
→unsatなパスをプロパティ違反（バグ）として報告
→unsatなパスを増やしてより効率的に
 - 健全性とパフォーマンスのトレードオフ
 - 良いpreconditionとは
 - 具体的すぎず（具体的すぎると探索能力が低下）
 - 抽象的すぎず（抽象的すぎるとパフォーマンスが低下）
 - 一般的なpreconditionのタイプ（対配列の例）
 - known-length : 配列長が分かっている
 - known-prefix : 配列内の（先頭から）一部が分かっている
 - fully known : 配列内の全要素の具体値が分かっている

5. PATH EXPLOSION

5.5 EXPLOITING PRECONDITIONS AND INPUT FEATURES

- preconditionによるパス探索の削減
 - 例: パケットのヘッダチェック
 - pkt: シンボル, header: 固定値を期待

```
1. start: get_input(&pkt);  
2.         for(k=0; k<128; k++)  
3.             if(pkt[k] != header[k])  
4.                 goto start;  
5.         parse_payload(pkt);
```

- A) headerの事前知識がない場合:
 - pktの先頭部分が探索されて3,4行目でパス爆発
- B) headerの128個目までの要素を知っている場合 (known-prefix) :
 - pktの先頭部分を固定できるので、ループ探索のスルーが可能

5. PATH EXPLOSION

5.6 STATE MERGING

- 状態のマージ

- 異なるパスを一つにまとめる強力な技術

$$\left. \begin{array}{l} S_1 = (stmt, \sigma_1, \pi_1) \\ S_2 = (stmt, \sigma_2, \pi_2) \end{array} \right\} S' = (stmt, \sigma', \pi_1 \vee \pi_2)$$

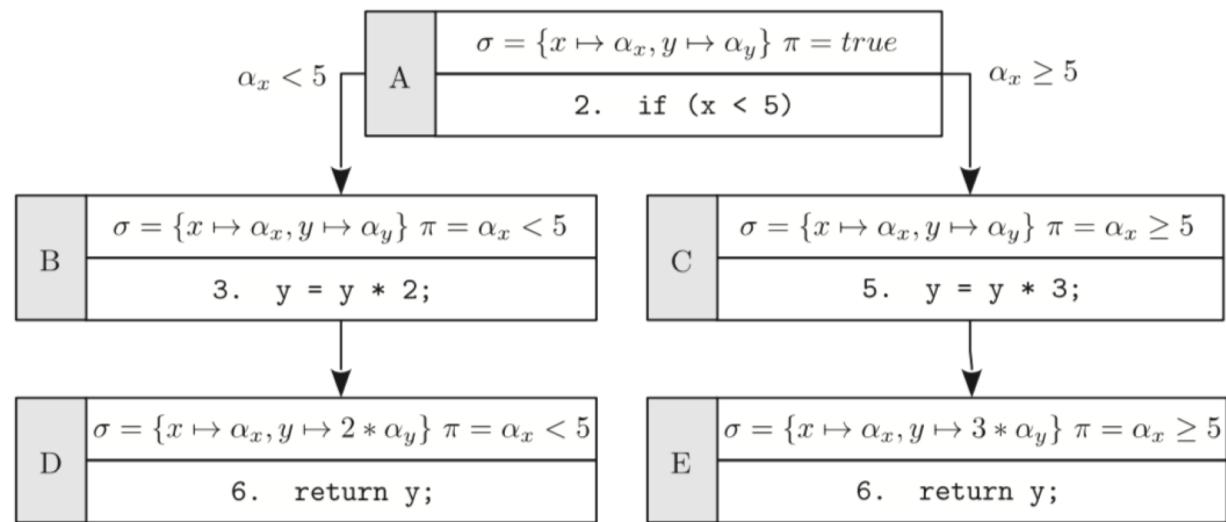
- 次ページで例題

5. PATH EXPLOSION

5.6 STATE MERGING

- 例題(マージなし)

```
1. void foo(int x, int y){
2.   if(x < 5)
3.     y = y * 2;
4.   else
5.     y = y * 3;
6.   return y;
7.}
```



(a)

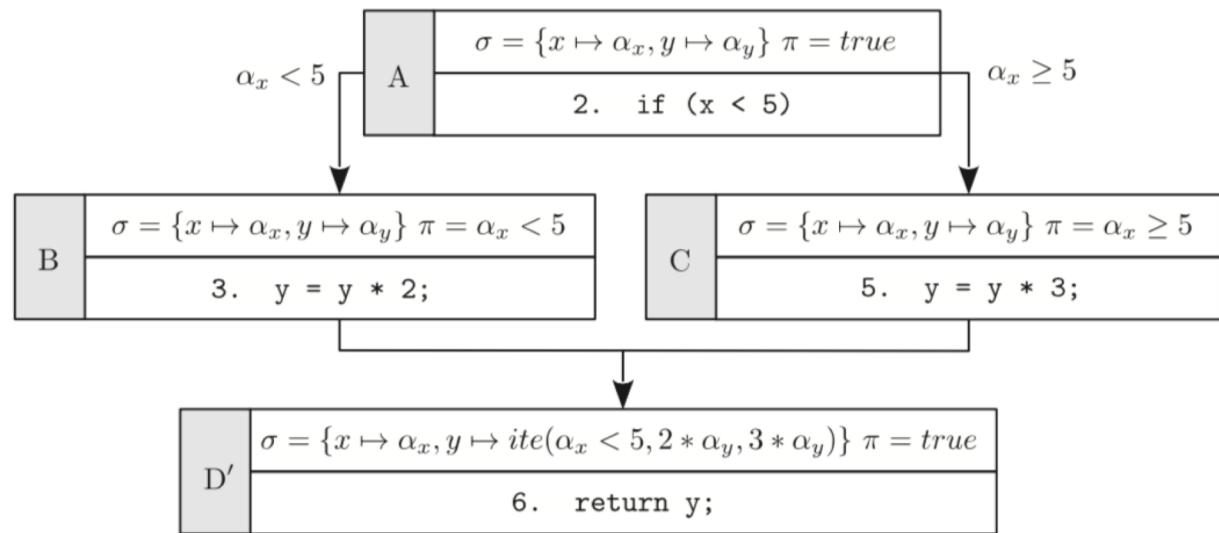
5. PATH EXPLOSION

5.6 STATE MERGING

- 例題(マージ)

```

1. void foo(int x, int y){
2.   if(x < 5)
3.     y = y * 2;
4.   else
5.     y = y * 3;
6.   return y;
7.}
  
```



- D' = (stmt, σ' , π')
 - σ' : yについてite式でマージ
 - $\pi' = \pi(D) \vee \pi(E) = \alpha_x < 5 \vee \alpha_x \leq 5 = \text{true}$

5. PATH EXPLOSION

5.6 STATE MERGING

- 状態のマージ

- 異なるパスを一つにまとめる強力な技術

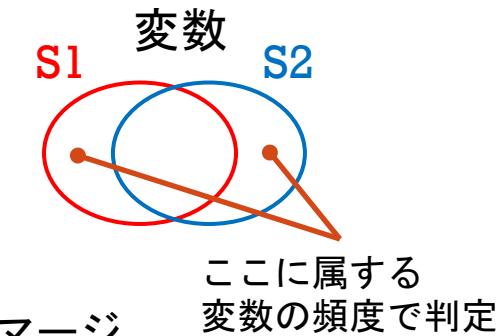
$$\left. \begin{array}{l} S_1 = (stmt, \sigma_1, \pi_1) \\ S_2 = (stmt, \sigma_2, \pi_2) \end{array} \right\} S' = (stmt, \sigma', \pi_1 \vee \pi_2)$$

- 利点: パス探索数を効率的に減らせる
 - 欠点: 制約を解くソルバの負担が増加し得る
 - 例題のように簡易化する場合もある
 - ソルバの処理時間が悪化すると結果的に非効率になる可能性もある

5. PATH EXPLOSION

5.6 STATE MERGING

- マージのヒューリスティクス
 - 高速化に寄与するマージ可能状態の探索
 - クエリカウント推定[70]
 - 各変数の分岐条件の使用頻度を推定
 - 2状態の相異なる変数が後の分岐で使用頻度が低ければマージ



- Veritesting[9]
 - 解析が簡単なステートメントのシーケンスに対して静的マージ(ite式)
 - 解析が困難なステートメントに遭遇したら記号探索
 - 解析が困難: 間接ジャンプ、システムコールなど

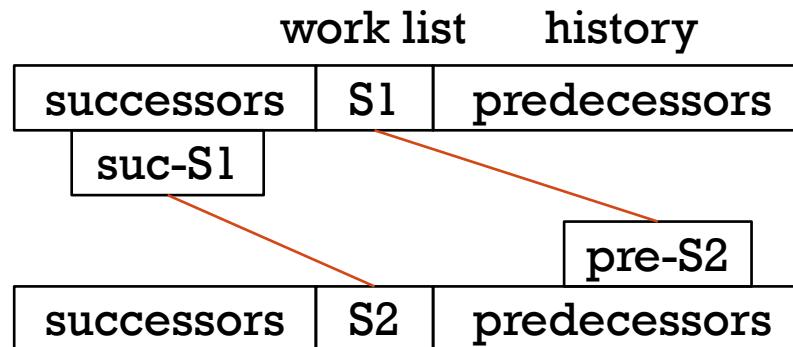
[70] Efficient state merging in symbolic execution [PLDI'12]

[9] Enhancing symbolic execution with veritesting [ICSE'14]

5. PATH EXPLOSION

5.6 STATE MERGING

- マージのためには、2状態の結果を把握している必要あり
 - →探索順序によってはマージの機会を逸する可能性
- 動的状態マージ (Dynamic State Merging[70])
 - エンジンは
 - work listにある探索候補の状態と、有限長のその前駆(predecessors): history を持つ
 - work list内の状態S1, S2をチェックする
 - pre-S2とS1, S2とsuc-S1の類似性が高いなら、S2とsuc-S1のマージを試行



5. PATH EXPLOSION

5.7 LEVERAGING PROGRAM ANALYSIS AND OPTIMIZATION TECHNIQUES

- エンジンの最適化に役立つ技術（ざっくりと）
- プログラムスライシング(program slicing)[110]:
 - プログラムの挙動の一部を表現する最小命令列を探索する
 - [44]: 逆方向にスライシング、ターゲットポイントへの誘導に利用
- 汚染解析(taint analysis):
 - ユーザ入力などの特定の入力の伝播追跡も可能
- Fuzzing:
 - ハイブリッドファジングへの応用[73, 101]
- 分岐予測(branch prediction):
 - 小さい幅のジャンプを避けることで、パイプライン実行の誤予測の障害を軽減

[110] Program Slicing [IEEE/TSE'84]

[44] Effective, static detection of race conditions and deadlocks [SOSP'03]

[73] Hybrid concolic testing [ICSE'07]

[101] Driller: Augmenting fuzzing through selective symbolic execution [NDSS'16]

5. PATH EXPLOSION

5.7 LEVERAGING PROGRAM ANALYSIS AND OPTIMIZATION TECHNIQUES

- 型チェック(type checking)
 - 記号実行にはできない型識別を組合せてより効率的なパス探索
- プログラム差分(program difference)
 - コード編集の影響を受けるデータフローの識別（依存解析）
 - Directed Incremental Sym Exe[116]:
影響を受けるノードの未探索領域をパス探索
- コンパイラ最適化(compiler optimization)
 - 一般にコンパイル最適化が記号実行に与える影響は不明確
 - ソルバ内部がブラックボックス扱いになっている節がある
 - 研究自体も少ない