C 言語ベース設計に対する高位設計検証技術

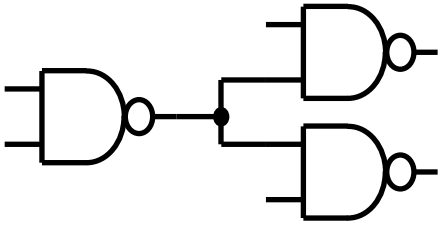
東京大学
大規模集積システム設計教育研究センター
藤田昌宏

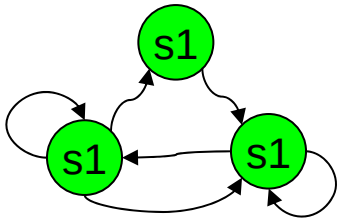
発表内容

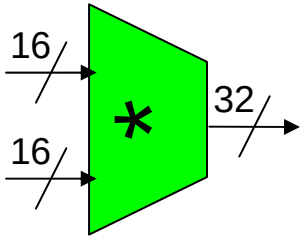
- SoC／組込みシステムの設計の流れと形式的検証
- 形式的検証技術の基礎
- モデルチェッキング／スタティックチェッキング
- 等価性検証
- まとめ

設計の各段階における利用される変数・関数

- ゲートレベル
  - AND, OR, …ゲートからなるネットリスト
    - Boolean関数・変数（2値）
- Register Transfer Level (RTL)
  - 各クロックサイクルごとの動作を記述 (FSM)
    - 多くの場合 Boolean変数
  - 掛け算器などの演算器の導入
    - ワード変数（固定長ビット, a[0:15]）による算術関数
- 高位・システムレベル
  - コントロールフロー制御文
    - Boolean変数とワード変数
  - 計算文や代入文
    - 多くはワード変数





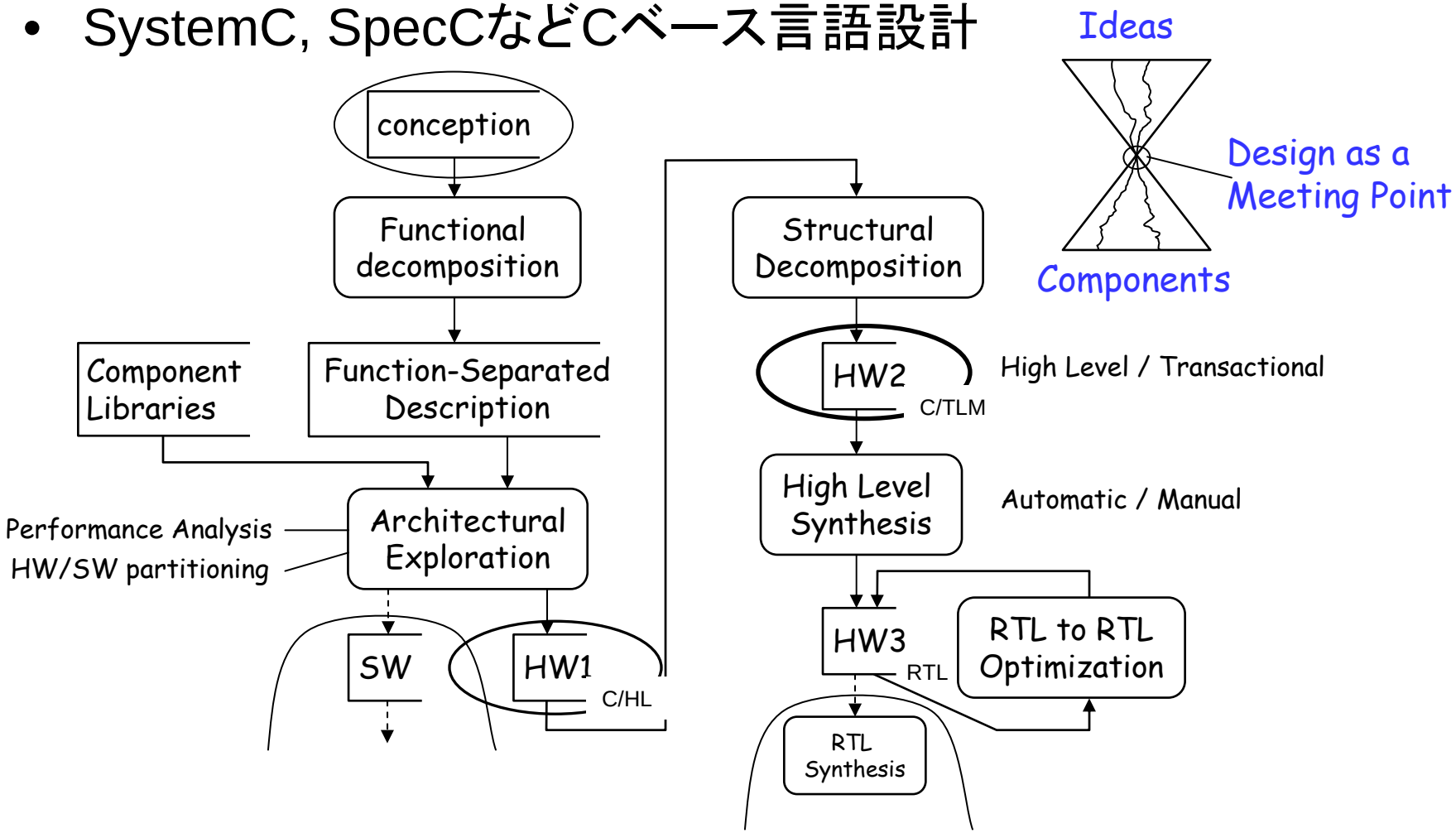



```
while (a>b) {…}
If (c==1) {…}
```

```
int a, b, c, x;
x = a + b * c;
```

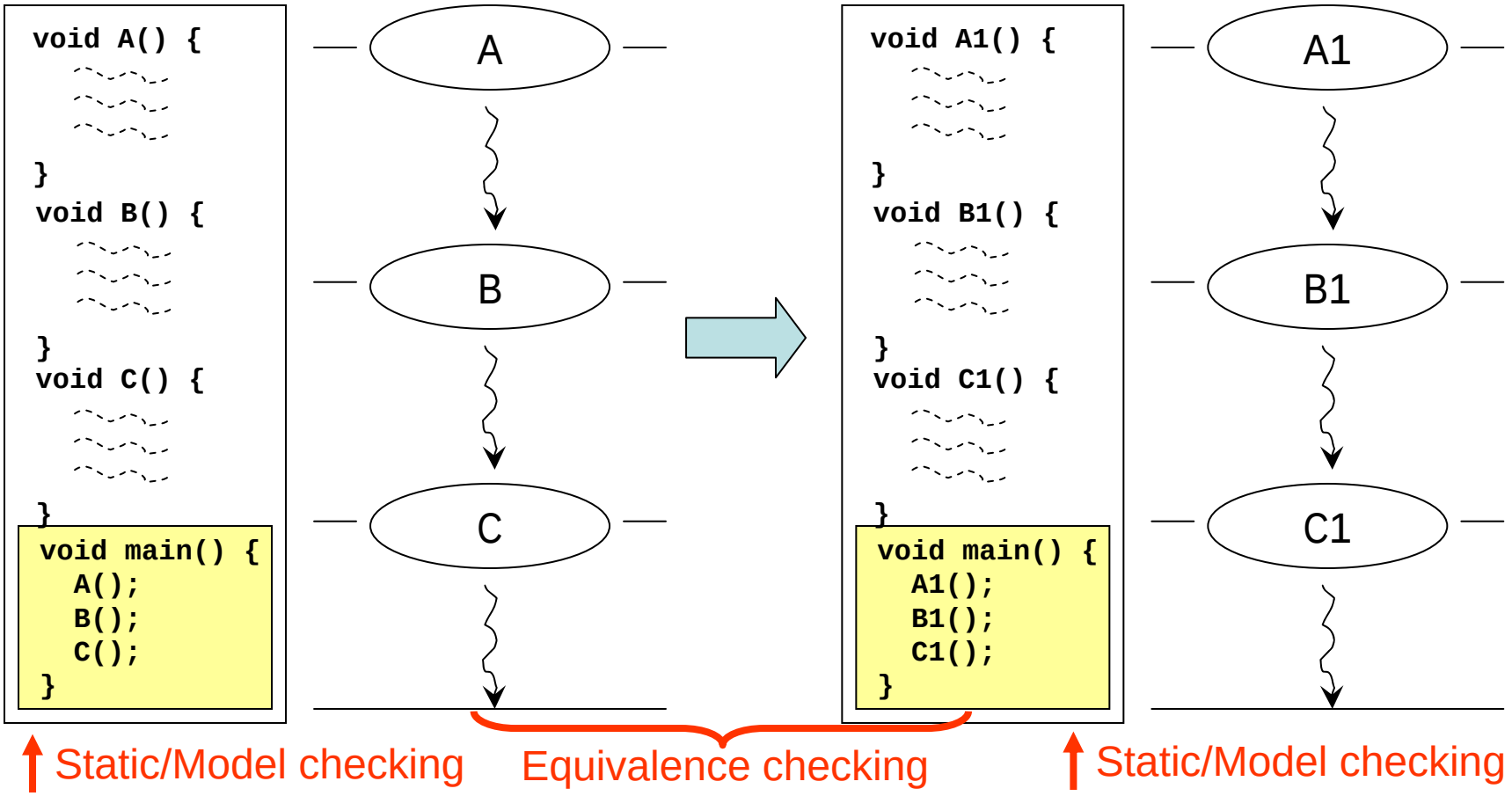
上位レベル設計の流れ

- 組込みシステム・SoC設計
- 企業で現在利用されている一般的な設計の流れ
- SystemC, SpecCなどCベース言語設計



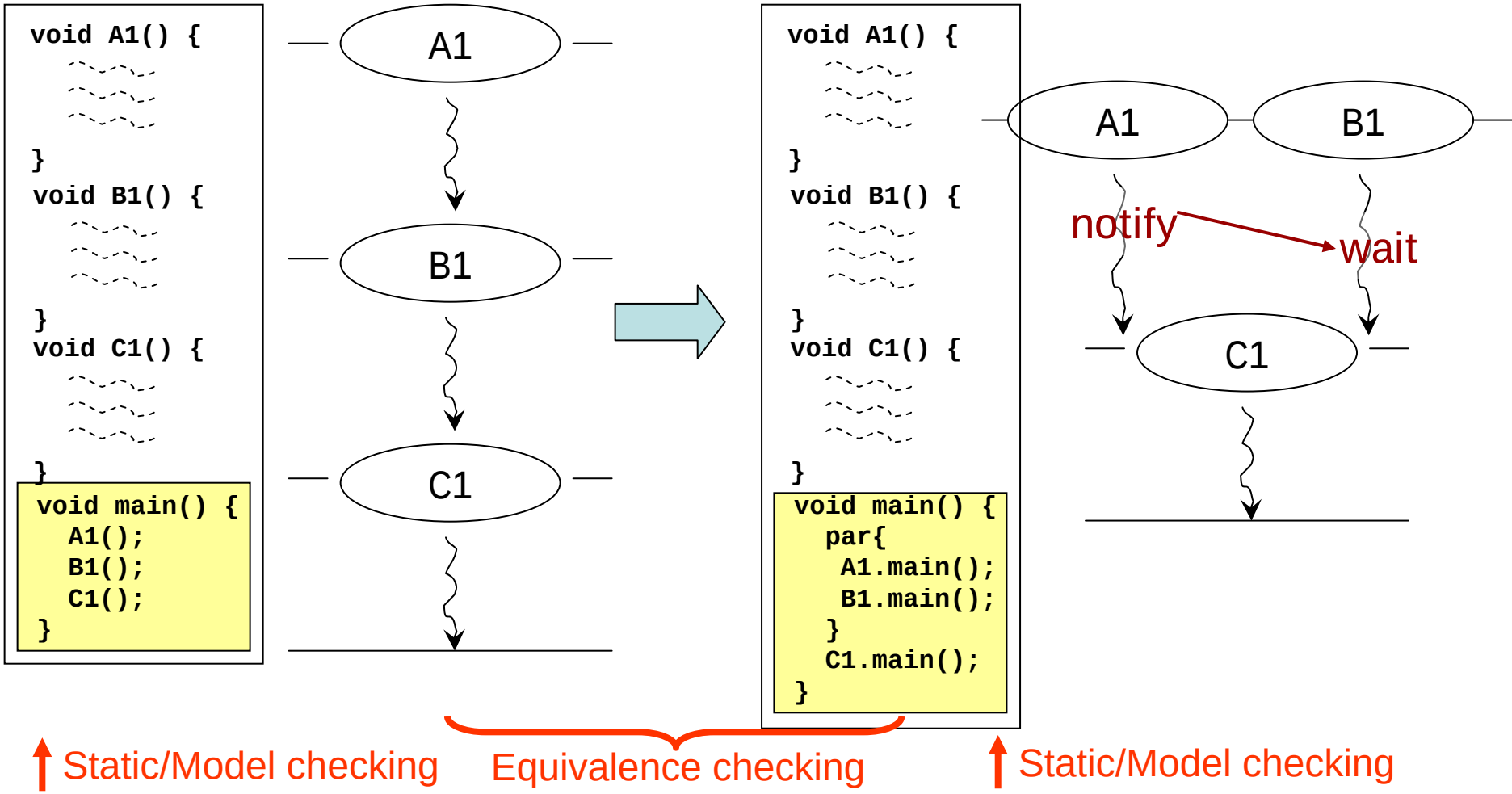
段階的詳細化(1)

- 機能を表現する記述から出発し、徐々に構造を導入するとともに、実行の並列化などを行っていく
  - A1, B1, C1 を A, B, C の詳細設計化



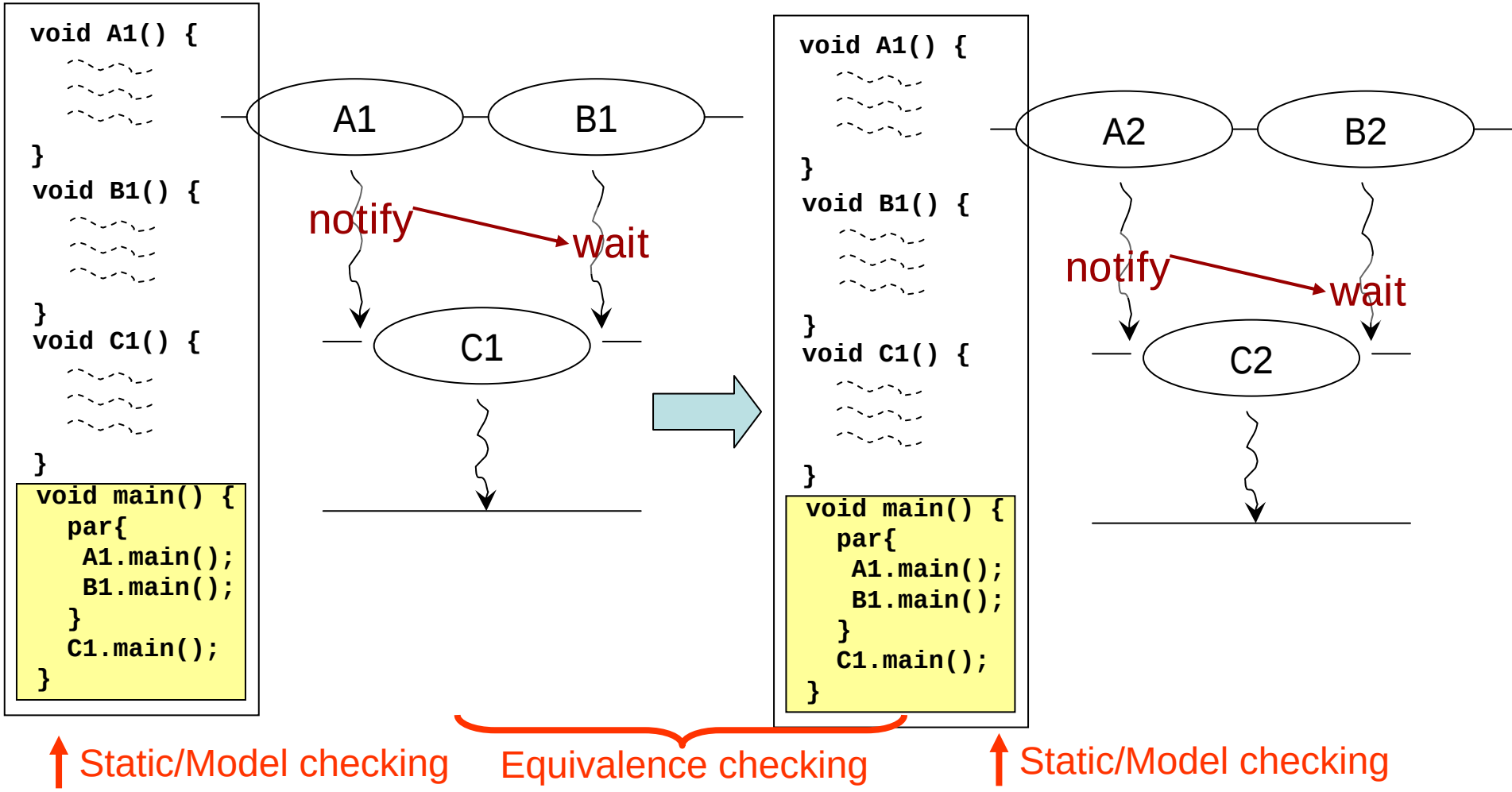
段階的詳細化(2)

- Change of control structures
  - IF-THEN-ELSE
  - Sequential to parallel



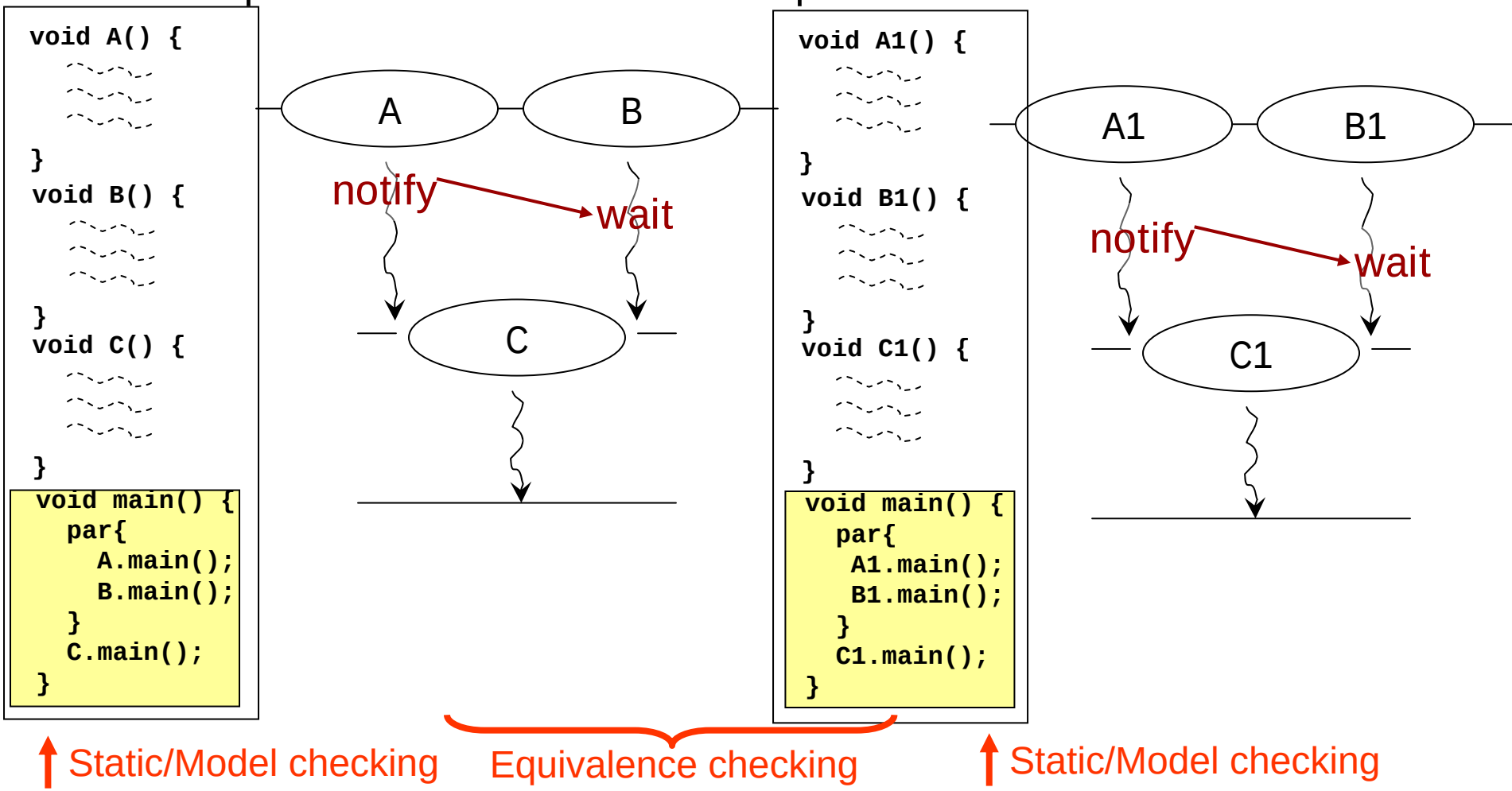
段階的詳細化(3)

- Again make each part more detailed
  - A2, B2, and C2 are refinement of A1, B1, and C1 respectively



段階的詳細化(4)

- If A and A1, B and B1, C and C1 are equivalent, then entire descriptions are equivalent
  - Comparison of two “similar” descriptions



設計詳細化のための各種変換

- アルゴリズムの変更
- 数式 → 浮動小数点 → 固定小数点
- 定数伝播、到達不能コード除去などのコンパイラにおける最適化など
- データフローグラフの最適化
  - 各種変換（実行順変更）
  - Loop unrolling
- 高位合成
  - Scheduling
  - Allocation/Binding
- 処理のパイプライン化（含むループ融合）
- メモリシステムの設計（階層化、キャッシュ）
- インターフェイスの詳細化、通信プロトコルに応じた変更
- Retiming

数式の簡単化・データフローグラフ変換

- Quartic-spline polynomial (3-D graphics)

  $P = zu^4 + 4avu^3 + 6bu^2v^2 + 4uv^3w + qv^4$

  (掛け算数 23)　　式を展開すれば分かる

- Horner form

  $P = zu^4 + (4au^3 + (6bu^2 + (4uw + qv)v)v)v$

  (掛け算数 17)　　式を展開すれば分かる

- 共通項の括りだし

  $d_1 = v^2$; $d_2 = d_1 * v$

  $P = u^3(uz + ad_2) + d_1(qd_1 + u(wd_2 + 6bu))$

  (掛け算数 11)　　式を展開すれば分かる

- 固定小数点演算によるオーバーフローの無視
  - 4ビット幅とする（modulo 2^4）

    $(a[0:3])^2 + 15(b[0:3])^2 = (a[0:3])^2 + 15(b[0:3])^2 - 16(b[0:3])^2$
    $= (a[0:3])^2 - (b[0:3])^2 = (a[0:3] + b[0:3])\ (a[0:3] - b[0:3])$

  Modulo演算に対する等価性の考慮が必要

  下位ビットではなく、上位ビットを使用する場合、クリッピング等もある

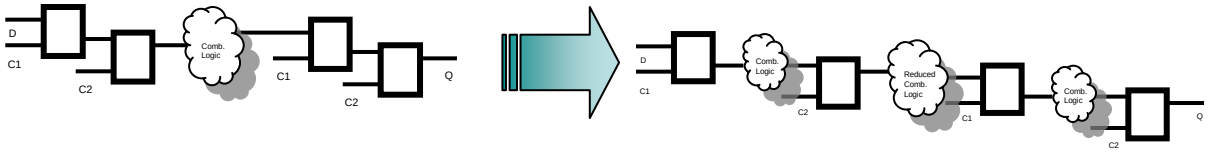
スカラー倍

- スカラー倍計算をする場合は多い
  - 4x + 5y + 6
  - Affine 変換
- しかし、掛け算器は回路規模が大きい
  - 32ビット掛け算器には100Kトランジスタ以上必要
- スカラー積は “shift” と “addition”で実現される
  - 4x = x<<2
  - 5y = 4y + y = y<<2 + y
  - 上の2つは一般の整数では等価だが、有限ビット幅では演算のビット幅、オーバーフローの扱いの管理が必要

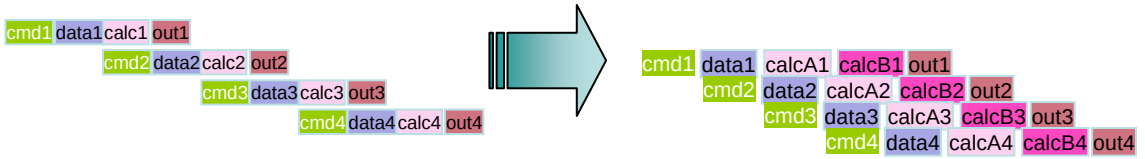
等価性ルールを設定して利用

各種変換(実行順も変化することも)

Re-timing
• 組合せ回路をラッチを越えて移動

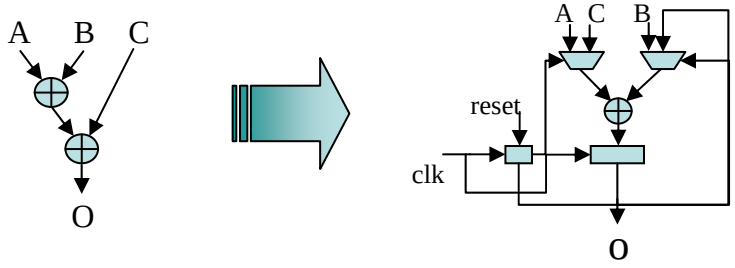


Pipelining
• Latency と throughputが変化



スケジューリング
• 実行ステップが増加

演算器などの共有
• 状態数やlatencyが変化

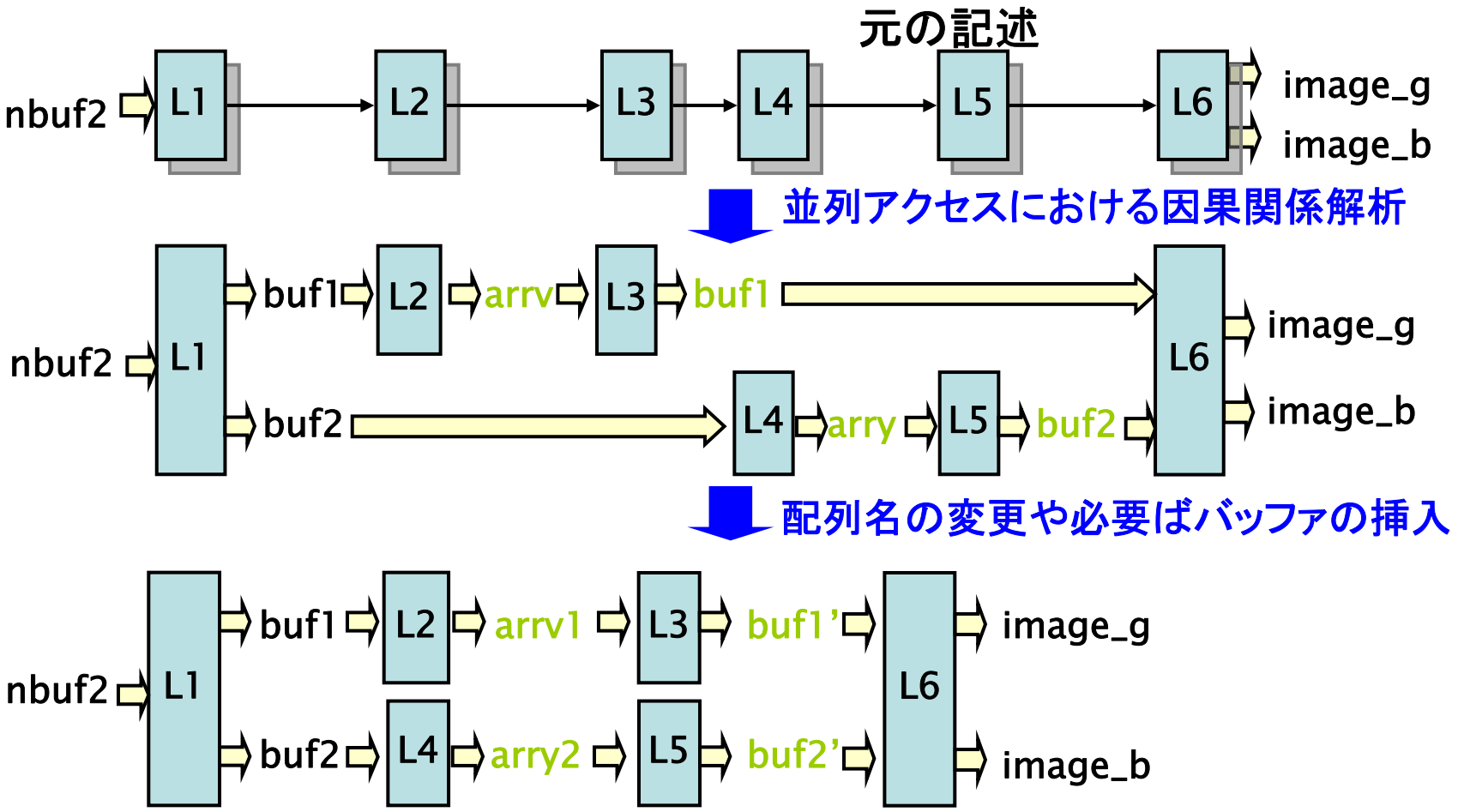


高位合成(動作合成)処理や人手最適化に伴って、上記のように変化する

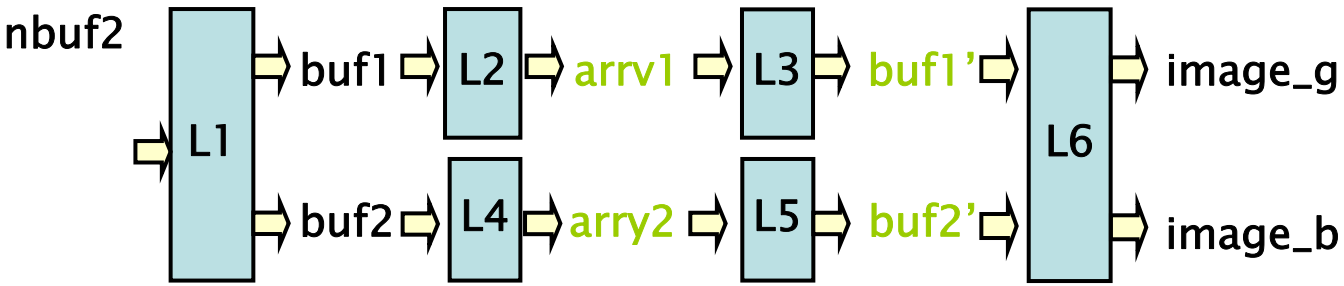
記号シミュレーションベースが一般的

パイプライン化のためのループ融合(1)

- パイプライン化による高速実行
  - 主に、C言語における “for-loop” 記述が対象
  - しかし、一般に個々のループは短いことも多い
  - ループを融合してより大きなパイプライン化を図る（並列度の向上）

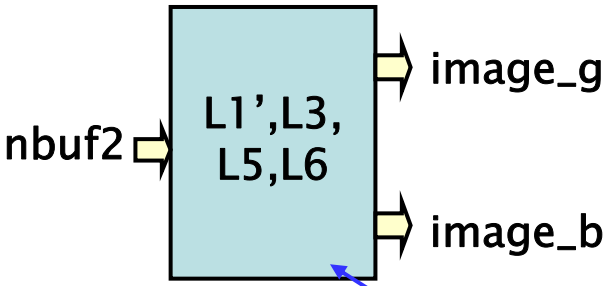


パイプライン化のためのループ融合(2)



ループ融合
- loop alignment
- loop peeling

scalar replacement



1つのループになれば、既存の自動パイプライン化技術が利用可能

例題: 画像処理で60段パイプライン化

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)     // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 100; i < 9899; i++)   // LOOP L2
6      c[i] = b[i-100] - 2*b[i] + b[i+100];
7  }
```

Loop alignment

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)     // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 200; i < 9999; i++)   // LOOP L2
6      c[i-100] = b[i-200] - 2*b[i-100] + b[i];
7  }
```

Loop guarding

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)   // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 1; i < 9999; i++)   // LOOP L2
6      if (i >= 200)
7        c[i-100] = b[i-200] - 2*b[i-100] + b[i];
8  }
```

Loop fusion

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++) {   // LOOP L1+L2
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5      if (i>=200)
6        c[i-100] = b[i-200] - 2*b[i-100] + b[i];
7    }
8  }
```

scalar replacement

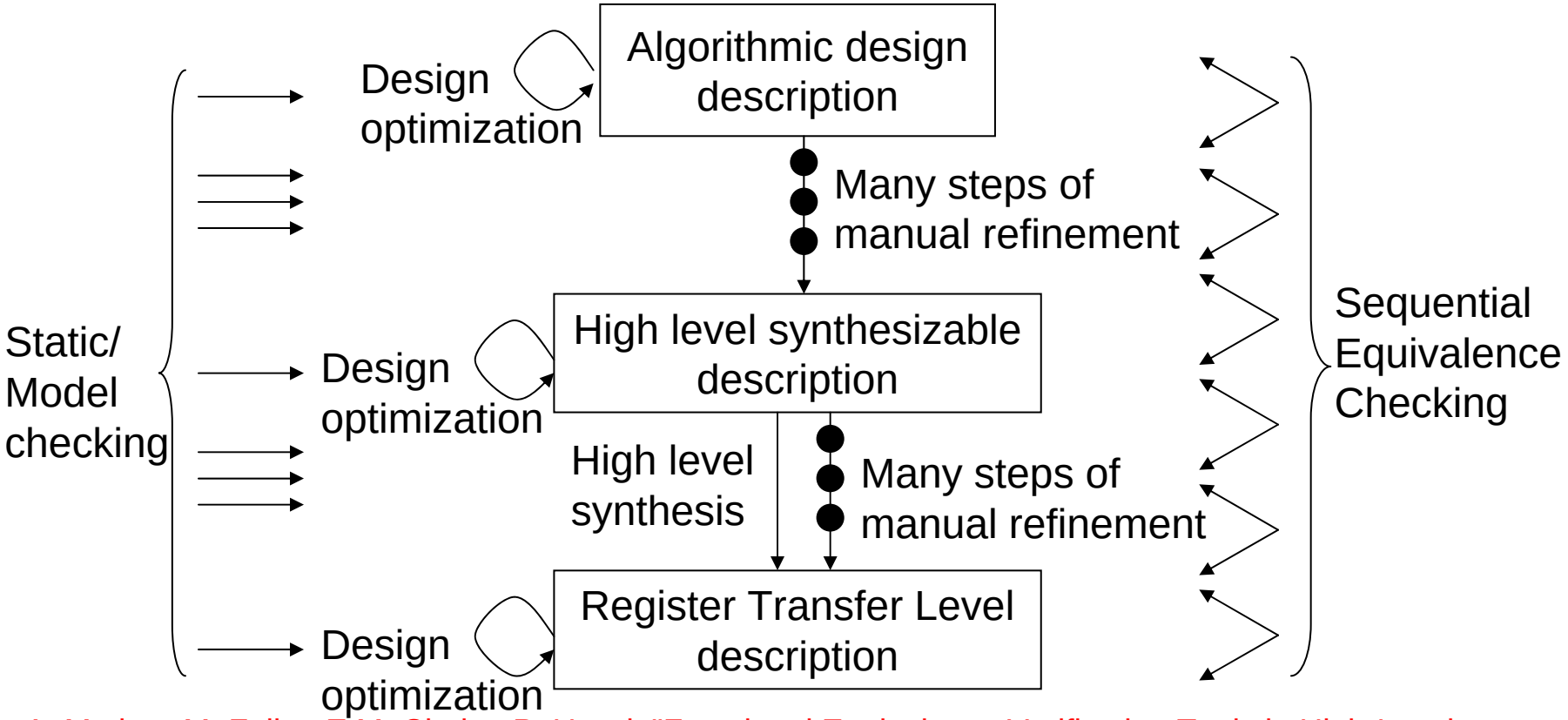
```
1  void filter(int a[], int c[]) {
2    int b[10000], i, tb1, ta1, ta2, ta3;
3    ta1 = a[0];
4    ta2 = a[1];
5    for (i = 1; i < 9999; i++) { // LOOP L1+L2
6      ta3 = a[i+1];
7      tb1 = ta1 - 2*ta2 + ta3;
8      b[i] = tb1;
9      if (i>=200)
10       c[i-100] = b[i-200] - 2*b[i-100] + tb1;
11     ta1 = ta2;
12     ta2 = ta3;
13   }
14 }
```

ループ処理専用の形式的解析手法が必要

イタレータや配列インデックスに対する依存関係を解析し、必要に応じて帰納法を利用

高位設計と検証問題

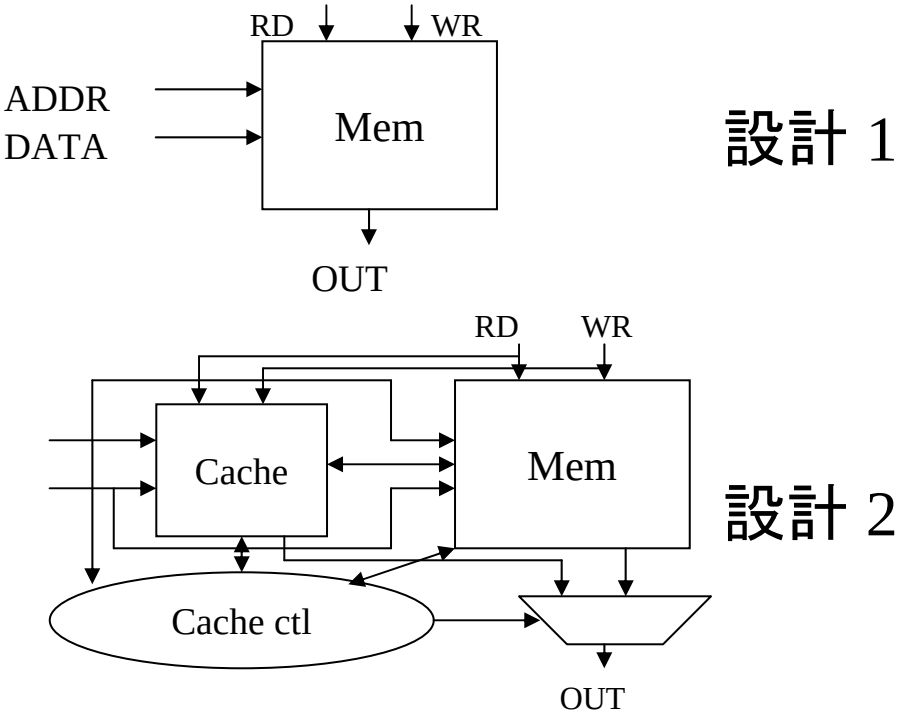
- 設計の各段階で、できるだけバグを混入させない
- それぞれの設計記述をモデル／スタティックチェッキング
- 2つの設計記述間の等価性検証
- シミュレーションだけでなく、形式的手法も導入



A. Mathur, M. Fujita, E.M. Clarke, P. Urard, "Functional Equivalence Verification Tools in High-Level Synthesis Flows," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 88-95, July/August, 2009.

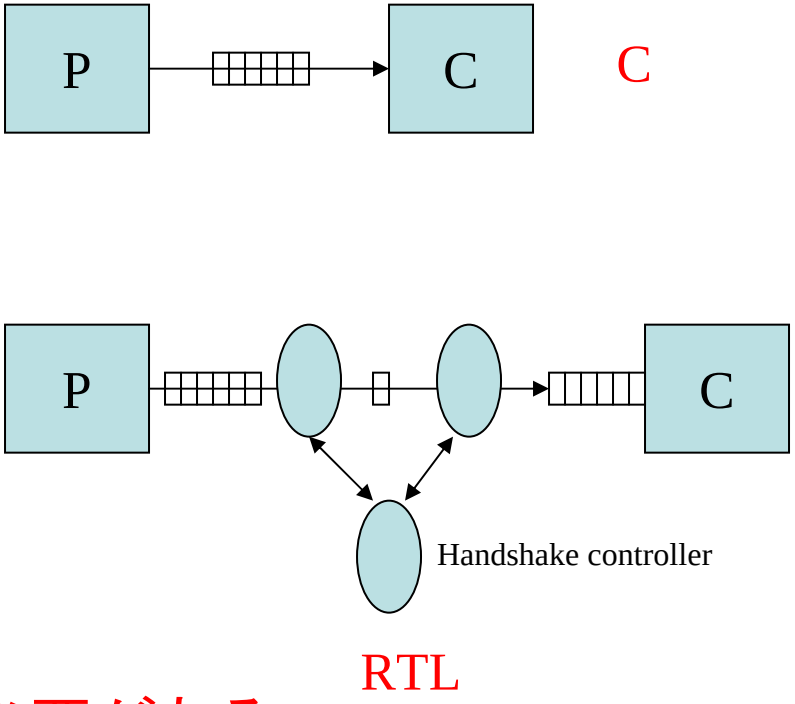
上位設計と下位設計の違いの例

- メモリ
  - 上位では、単なる配列へのアクセス
  - 下位（RTL）では、キャッシュ付きのメモリシステム
  - メモリに対する、read, writeなどは1つのトランザクションとなる



下位設計にのみ現れるもの

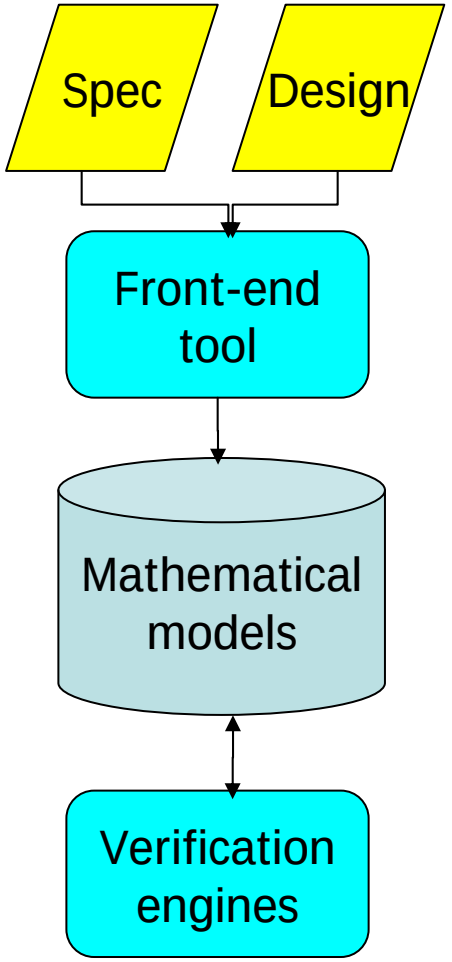
- RTL以下の設計では下記も含んでいる
  - 故障テスト用スキャンチェイン
  - スリープモード用論理
  - Clock gating
  - キャッシュメモリ
  - バス arbitrationを含んだバス通信
  - ハンドシェイクなどの通信プロトコル



RTL以下において、検証する必要がある

形式的検証の流れ

- 設計の正しさを数学的に証明する
  - 設計も仕様も数学モデルに変換される
  - 数学的な推論
  - 全ての場合をシミュレーションすることと等価だが、全ての場合を調べる分けではない
- 利用される数学モデル
  - Boolean 関数（命題論理）
    - 計算機上で如何に効率よく操作できるか？
  - 1階述語論理、あるいはそのサブセット
    - ワード変数
    - 効率的に処理可能、かつ有意義なサブセット
  - 高階論理（主に定理証明システム）
    - 基本的に対話的（人が自分で考える）
- 言語フロント・エンドも非常に重要
  - 全体性能を決めてしまう場合も多い



解析手法

- Boolean 関数・変数
  - 基本的に2値（0/1）
  - Binary Decision Diagram (BDD) やその拡張
  - SAT ソルバー
- ワード変数やその関数
  - 無限ビット幅（Infinite precision）
    - Uninterpreted functions with equality やその拡張
  - 有限ビット幅 (固定ビット幅)
    - SMT ソルバー
    - 決定グラフ
    - modulo 演算に関する定理の利用

決定グラフ

- Boolean 関数 (f: $B^n \rightarrow B$)
  - Binary Decision Diagrams (BDD, FDD, KFDD, etc.)

$$f(x, y, \ldots) = \bar{x}.f(x=0, y, \ldots) + x.f(x=1, y, \ldots)$$
$$= \bar{x}.f_{\bar{x}} + x.f_x$$
$$= f_{\bar{x}} \oplus x.f_{\delta x} \qquad f_{\delta x} = f_x \oplus f_{\bar{x}}$$
$$= f_x \oplus \bar{x}.f_{\delta x}$$

- 算術関数 (f: $B^n \rightarrow Z$ (integer))
  - Multi-terminal (MTBDDs), Algebraic Decision Diagrams (ADDs)

$$f = x.f_x + (1-x).f_{\bar{x}}$$

  - Binary Moment Diagrams (*BMD, K*BMD, *PHDD)

$$f = f_{\bar{x}} + x.f_{\delta x}$$
$$f_{\delta x} = f_x - f_{\bar{x}}$$

掛け算に対して効率的

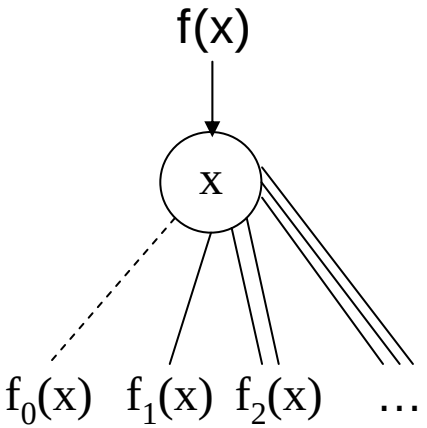
Taylor Expansion Diagram

- 算術関数 (f: Z → Z)
  - fを微分可能な連続関数とする
  - Taylor 展開:

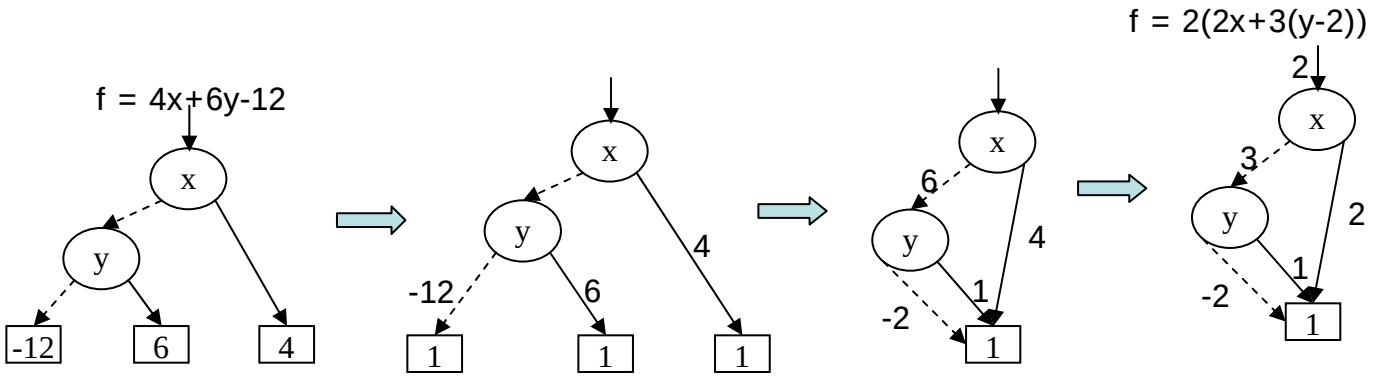
$$f(x,y,...) = f(x=0,y,...) + x.f'(x=0,y,...) + \frac{1}{2!}x^2.f''(x=0,y,...) + ...$$

  - 表記
    - $f_0(x) = f(x=0,y,\ldots)$ 0-child - - - - - -
    - $f_1(x) = f'(x=0,y,\ldots)$ 1-child ----------
    - $f_2(x) = ½*f''(x=0,y,\ldots)$ 2-child ======
    - etc.

$$f(x,y,...) = f_0(x) + x.f_1(x) + x^2.f_2(x) + ...$$



- Horner Expansion Diagram
  - 定数ノードは2つ: 0 と 1
  - エッジに互いにrelatively primeになるようにウェイトを付加し、共有により正規形とする



Satisfiability (SAT、充足可能性判定) 問題

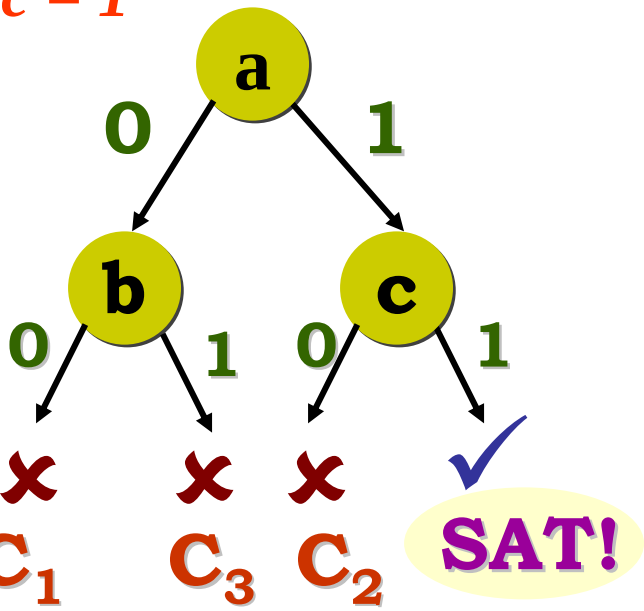
f がCNF 式で与えられる :

- 変数の集合 : V (a,b,c)
- クローズの積 (C_1,C_2,C_3)
- 各クローズはリテラルの和

f を1とするような、各変数への値の割り当ては存在するか？

存在するならその例を示し、存在しないならそれを証明する

例 : $\underbrace{(a+b)}_{C_1}\underbrace{(a+c)}_{C_2}\underbrace{(a+b)}_{C_3}$ $a = c = 1$

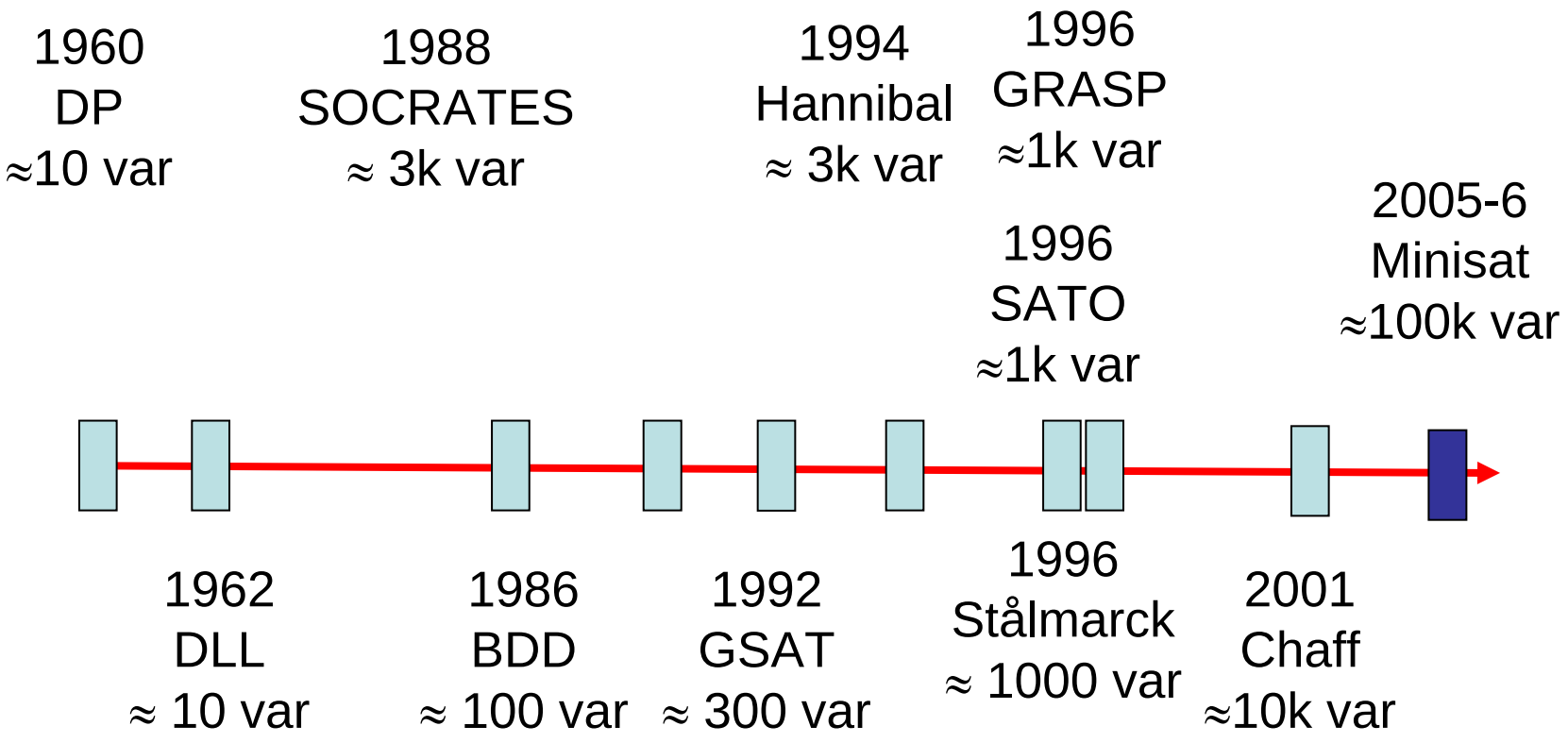


DPLL アルゴリズム

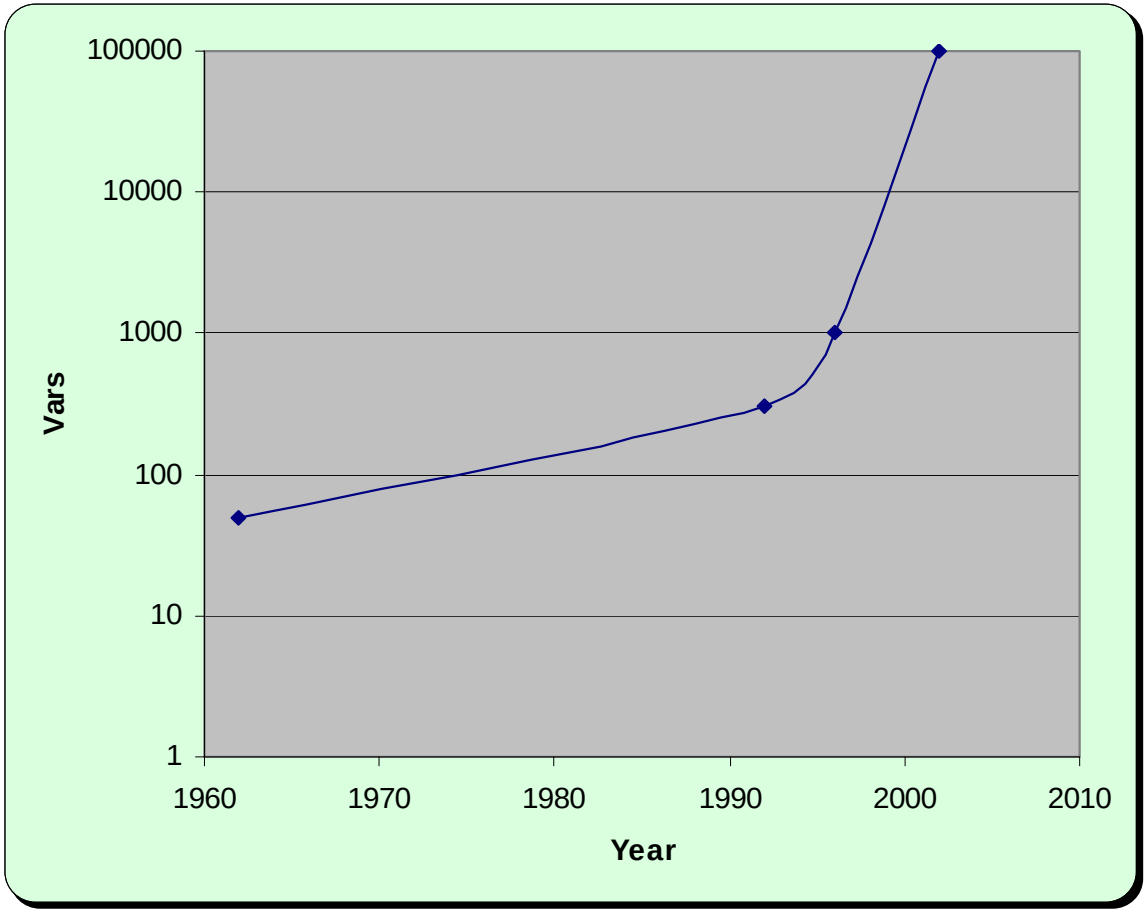
CNF 式 f(v1,v2,..,vk)と 場合分けの変数決定法が与えられる

充足するまで、場合分けとバックトラックを繰り返す

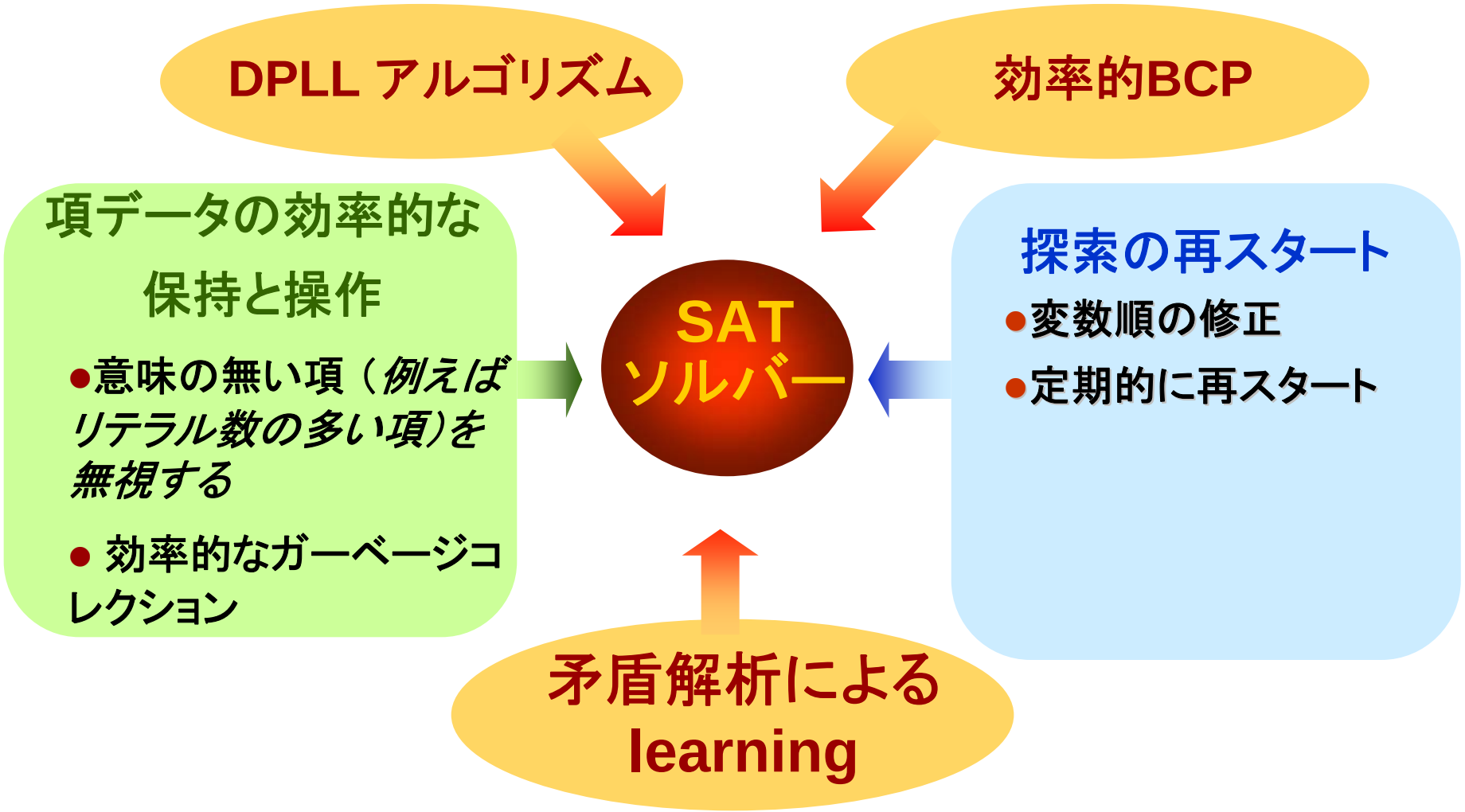
SAT手法の歴史


1960
DP
≈10 var
1962
DLL
≈ 10 var
1988
SOCRATES
≈ 3k var
1986
BDD
≈ 100 var
1994
Hannibal
≈ 3k var
1992
GSAT
≈ 300 var
1996
GRASP
≈1k var
1996
SATO
≈1k var
1996
Stålmarck
≈ 1000 var
2001
Chaff
≈10k var
2005-6
Minisat
≈100k var

SAT 手法は、最近、急速に進歩している！



最新の SAT ソルバー

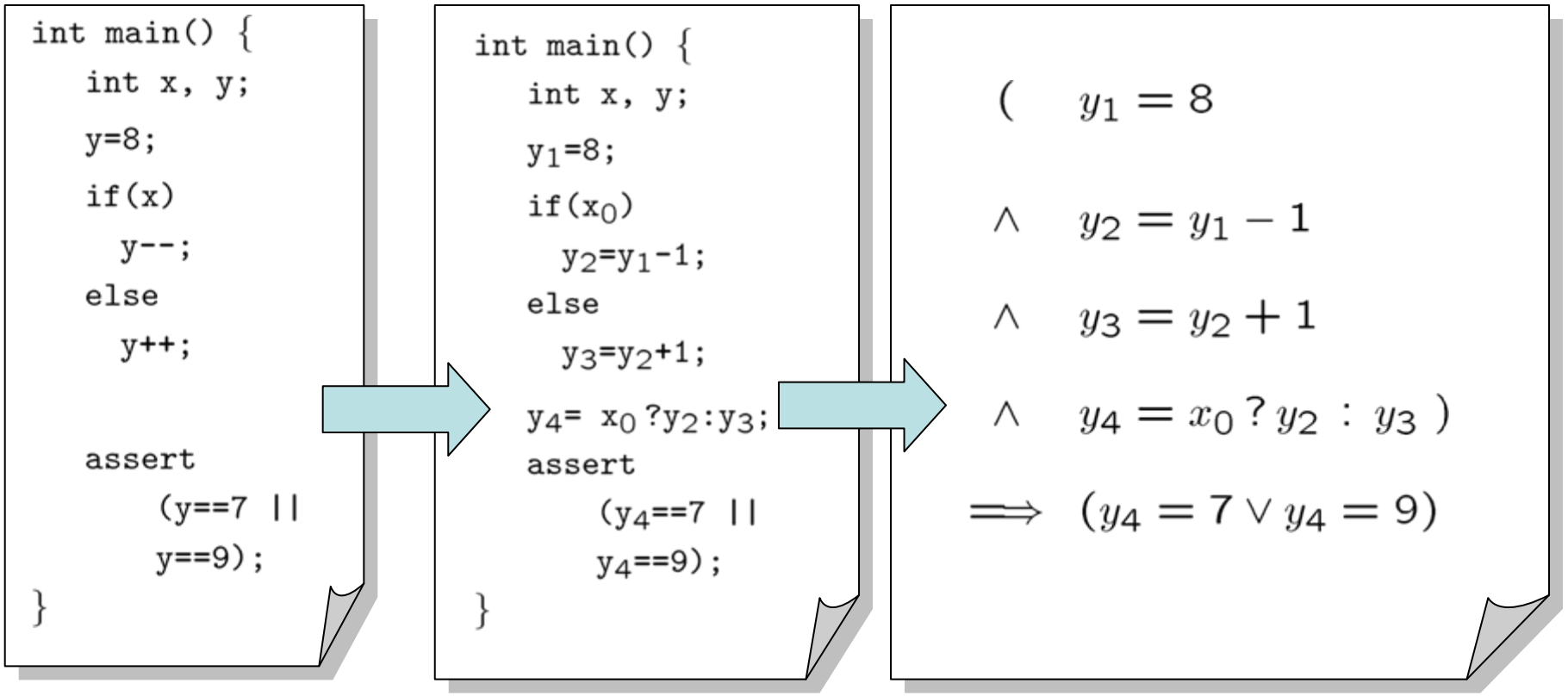

DPLL アルゴリズム
効率的BCP
項データの効率的な保持と操作
●意味の無い項（*例えばリテラル数の多い項）を無視する*
● 効率的なガーベージコレクション
SAT ソルバー
探索の再スタート
●変数順の修正
●定期的に再スタート
矛盾解析による learning

SATツールの性能

- 最近、性能が飛躍的に向上
  - Conflict clause learning (GRASP, zChaff)
  - Conflict-driven variable ordering (zChaff, BerkMin)
- 性能
  - *数百万変数*で *数百万万項*程度の式の判定が *数時間* で行える
  - ただし、*構造的な問題*、つまり、実際の設計から生成された問題に限る
- ツールの開発状況
  - パブリックドメイン
    - GRASP : Univ. of Michigan
    - SATO: Univ. of Iowa
    - zChaff: Princeton University
    - BerkMin: Cadence Berkeley Labs.
  - 商品
    - PROVER: Prover Technologies

C言語記述から論理式への変換

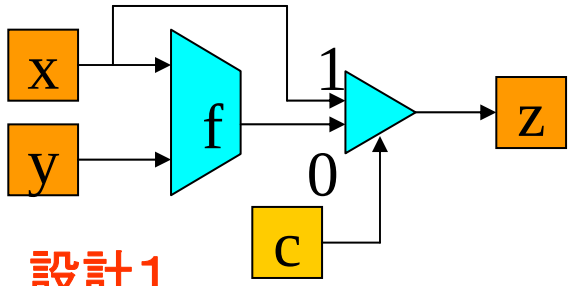
- 代入文ごとに新しい変数を導入
- if文などの条件を追加
- ループは、基本的に展開



ワード変数や算術演算をBooleanに展開しなければいけない
各代入ごとに新しい変数にしなければならない
→ SATソルバーで扱えるものは限られる

Uninterpreted Functions with Equality

- 算術演算は、記号として扱う



設計1

⇨ **z:= if c then x else f(x,y)**

設計2

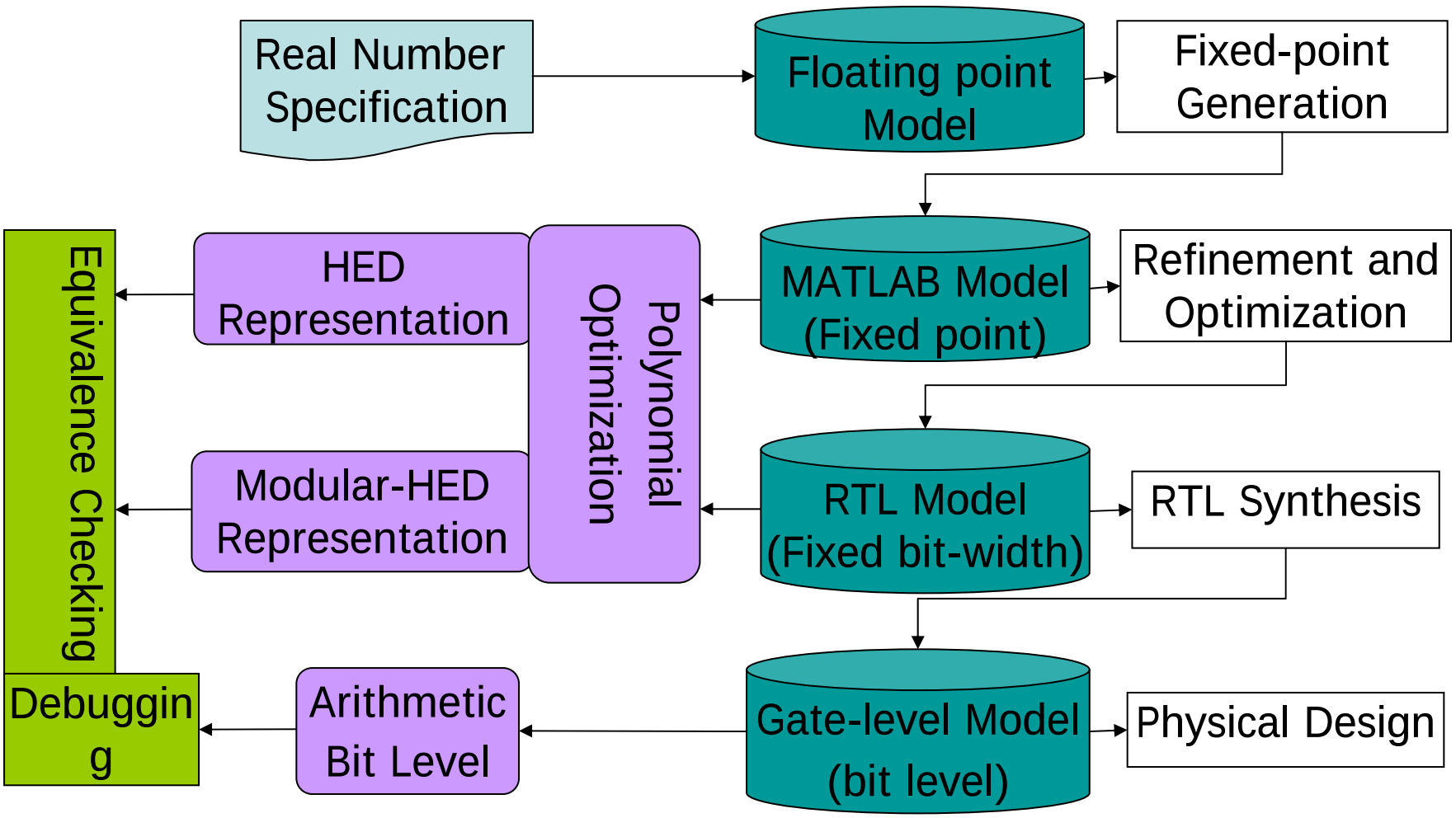
- 等価性検証：“Design1 = Design2 ?”
  - 決定手続きの利用による、自動検証が可能
  - 論理変数に展開すると
    - 各変数32ビットずつ必要
    - 算術演算 f によっては、複雑で処理不能（たとえば掛け算）
- 別の等価な例
  - 設計1：if a=b then X<-a*c else X<-b*c
  - 設計2：X<-b*c

このようなDecidableな問題を扱うソルバーを複数用意して切り替えるツールは、SMT（Satisfiavility Modulo Theory）ソルバーと呼ばれ、Z3, CVC3など多数開発されている

SMTソルバー ＝ 定理証明ツールの自動化できる部分を取り出したもの

Linear arithmetic, Inequality, Presburger logic, …

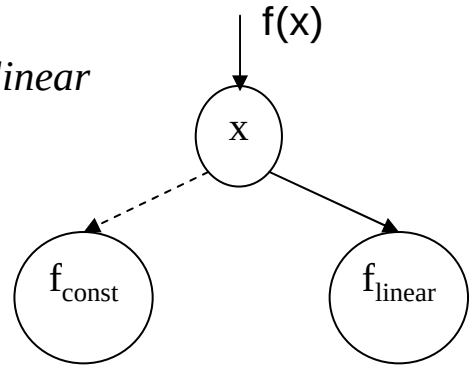
算術演算(DSP)向き検証技術



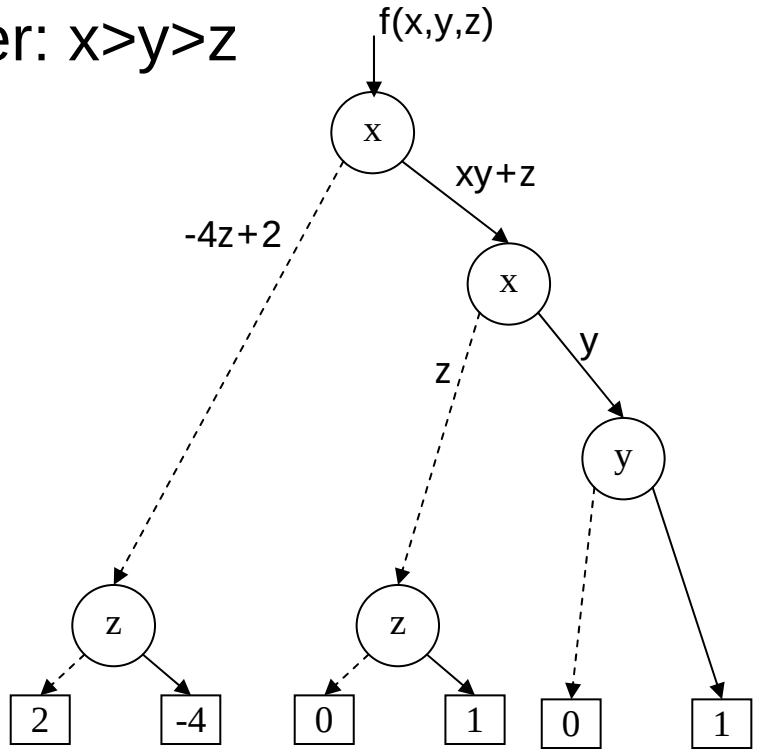
算術式を如何に効率的に取り扱えるか？

Horner Expansion Diagram (HED)

- Horner Expansion: $f(x, y, ...) = f_{const} + x.f_{linear}$
  - Const（左）エッジ（破線）
  - Linear（右）エッジ（実線）



- 例
  - $f(x,y,z) = x^2y+xz-4z+2$; Order: x>y>z
  - $f(x,y,z) = [-4z+2]+x[xy+z]$
    $= f_{const}+x.f_{linear}$
    - $f_{const} = -4z+2 = f(z)$
    - $f_{linear} = f_1(x,y,z) = xy+z$
  - $f_1(x,y,z) = xy+z = z+x[y]$
    - $f_{1const} = z$; $f_{1linear} = y$
  - Horner form
    - f = x(x(y)+z)+z(-4)+2

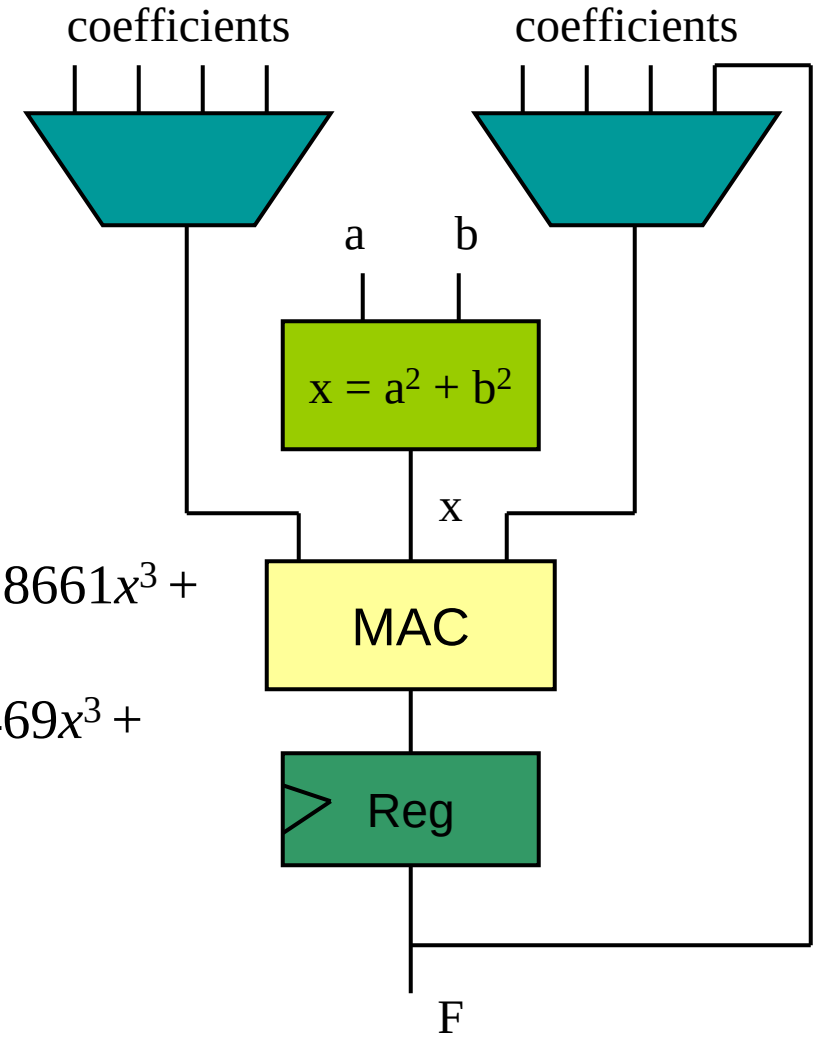


例題

- Anti-aliasing 関数 $F = 1/(2\sqrt{(a^2+b^2)}) = 1/(2\sqrt{x})$
- 一定次数までTaylor 展開

$$F \approx \frac{1}{64}x^6 - \frac{9}{32}x^5 + \frac{115}{64}x^4 - \frac{75}{16}x^3 + \frac{279}{64}x^2 - \frac{81}{32}x + \frac{85}{64}$$

- 固定ビット幅で実装
  - $F_1[15:0]$, $F_2[15:0]$, $x[15:0]$
  - $F_1 = 156x^6 + 62724x^5 + 17968x^4 + 18661x^3 + 43593\,x^2 + 40244x + 13281$
  - $F_2 = 156x^6 + 5380x^5 + 1584x^4 + 10469x^3 + 27209\,x^2 + 7456x + 13281$
  - $F_1 \neq F_2$ over Z
  - $F_1[15:0] = F_2[15:0]$



Smarandache 関数

- ***Smarandache 関数*** *S(b)*とは、正数でありその階乗 $S(b)!$ が *b* で割り切れるものの最小値
  - 例えば、1!, 2!, 3!, は8で割り切れないが、4! は割り切れるので(4!/8 = 3)、S(8) =4
    - 1*2*3*4 % 8 = 0
    - 5*6*7*8 % 8 = 0
    - 11*12*13*14 % 8 = 0
    - 100*101*102*103 % 8 = 0
  - 連続する 4 つの整数の積は8で割り切れる
    - **x(x+1)(x+2)(x+3) ≡ 0 mod 8**

Modular-HED (M-HED)

- **Vanishing Polynomial**:

$$g(x) = \prod_{i=1}^{S(n)} (x+i) \equiv 0 \bmod n$$

- もし与えられた多項式 g(x) をS(n) 個の連続する整数の積の形に変形できれば、その多項式は Z_n において 0 と等価
  - $f(x) = x^6 + 21x^5 + 175x^4 + 735x^3 + 1624x^2 + 1764x + 720$ over Z_{2^4}
  - $S(2^4) = 6$
    - $f(x) = (x+1)(x+2)(x+3)(x+4)(x+5)(x+6) \equiv 0 \bmod 16$
  - 従って、f(x)を削除できる
  - 任意の多項式に対し、f(x)を何回足しても、引いても modulo n の元では、変化しない
  - 多変数へ拡張して、等価性検証に利用
  - HEDなどの決定グラフを利用することで、複雑な多項式間の等価性を高速に判定可能

実験結果

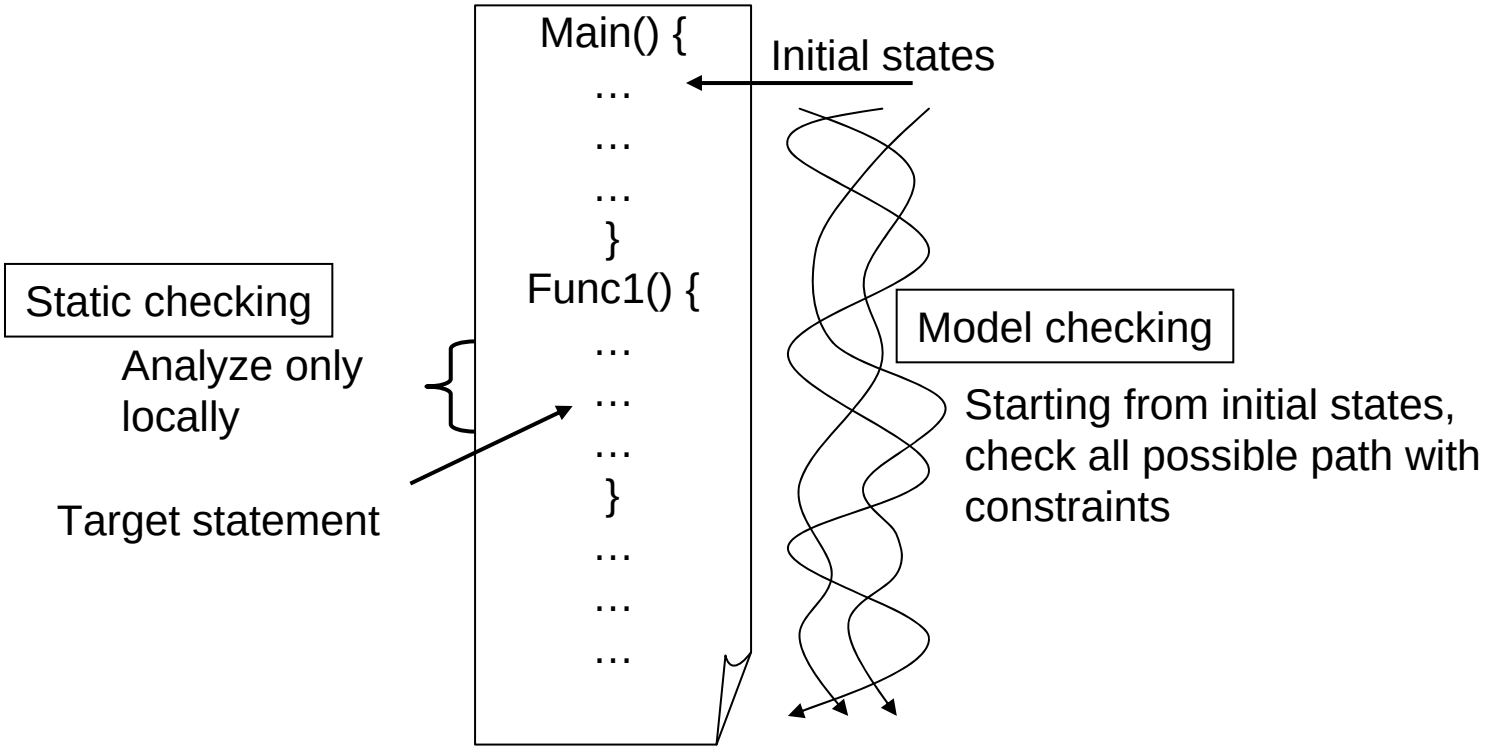
Benchmark	Specs	Modular-HED	Method [9]	CUDD-BDD	*BMD	miniSat [13]	MILP [12]
	Var/Deg/n	Node/Time	Time (s)	Node/Time	Node/Time	Vars/Clauses/Time	Time (s)
AAF Anti-aliasing function	1 / 6 / 16	8 / 0.016	6.81	1.1M / 32.2	NA / >500	3.9K / 107K / >500	>500
D4F Degree-4 filter	1 / 4 / 16	6 / 0.031	4.95	27M / 20.3	NA / >1000	25K / 76K / >1000	>1000
CHEB Chebyshev polynomial	1 / 5 / 16	7 / 0.01	5.95	1M / 26.9	NA / >500	3.5K / 86K / >500	>500
PSK Phase-shift keying	2 / 4 / 16	16 / 0.032	13.48	NA / >500	NA / >500	52K / 142K / >500	>500
DIRU Digital image rejection unit	2 / 4 / 16	9 / 0.016	14.4	NA / >1000	NA / >1000	10K / 30K / >1000	>1000
MI automotive applications	2 / 9 / 16	26 / 0.2	17.5	23M / 39.4	NA / >1000	24K / 69K / >1000	>1000
SG Savitzky Golay filter	5 / 3 / 16	35 / 0.24	6.1	NA / >1000	NA / >1000	64K / 190K / >1000	>1000
QS Quartic Spline polynomial	7 / 4 / 16	19 / 0.09	32.4	NA / >1000	NA / >1000	76K / 211K / >1000	>1000

NA: Not Applicable; K: Thousand; M: Million

ワードレベルツールの性能 >> Booleanレベルツールの性能

スタティックチェッキングとモデルチェッキング

- モデルチェッキングは初期状態（リセット状態）から出発し、到達可能な状態を網羅的に調べる（調べようとする）
  - しかし、任意の状態から解析を始めることも可能
  - 設計を一度抽象化してから、解析を始めることも可能
- スタティックチェッキングは検証項目に密接に関係した記述部分から解析を始め、かつ解析自体も部分的（ローカルな解析）
  - しかし、解析する範囲は拡張可能



状態空間探索

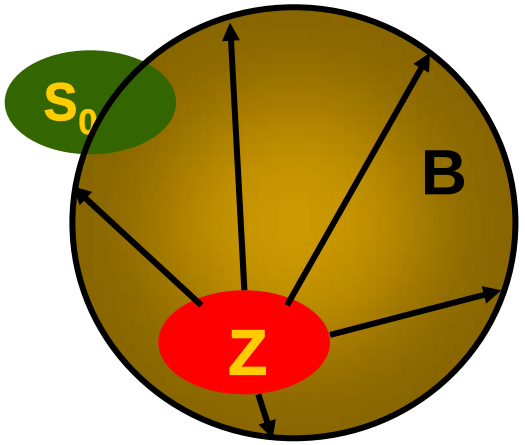
例: Safety プロパティ

S_0: 初期状態　　Z：悪い状態

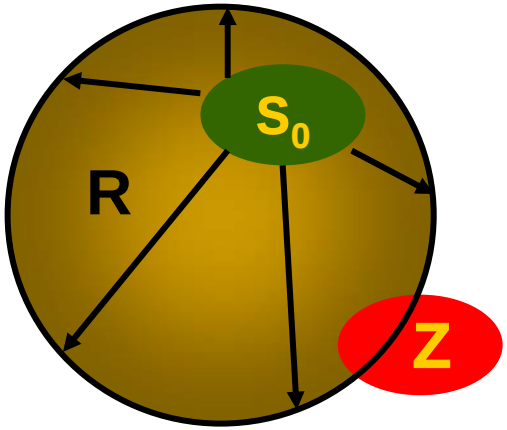
❶ S_0 (Z) からはじめる

❷ 状態遷移を順に前方(後方）に辿り、到達可能な状態の集合 R (B) を求める

❸ 集合R(B)の中に$Z(S_0)$があるか調べる



後方探索



前方探索

- 明示的プロパティ検証
  - 個々の状態を1つずつ辿っていく
  - 状態変数（FF）が多い場合は処理不能
  - Murphi, SPINなど実用的ツールが存在
- シンボリック・プロパティ検証
  - 状態空間を論理関数で表現し、状態の集合の処理で実現
  - 2分決定グラフや論理関数充足判定（SAT）手法を応用

モデルチェッキングアルゴリズム

- 状態を明示的に辿るアルゴリズム
  - 状態を1つ1つ明示的に辿ることで解析していく
  - 状態数が多い場合（記憶素子数が多い）、すべてを辿ることは不可能
  - しかし、一定の深さ（状態遷移列の長さ）までなら実用的: Murphi, SPIN, …
- 記号モデルチェッキング
  - 状態を辿る操作と到達した状態の集合を何らかの論理式で表現し、BDDやSATソルバーを使って計算していく
  - 処理手順:
    - 状態遷移と状態の集合を論理式で表現
    - 幅優先で状態遷移をForward/backward に辿っていく
    - 状態遷移の image/pre-imageを論理式として計算する
    - Fixed-point になるまで続ける

      メモリオーバーや計算時間の関係でfixed-pointに到達しない場合も多い
- 両者を組合せたもの
  - トレードオフ
    - 初期状態から、できるだけ遠くに到達したい
    - できるだけ網羅的に解析したい
- 初期状態からという条件を取り除く
  - スタティックチェッキングと基本的に同じ

バグ発見手法としてのモデル・スタティックチェッキング

- 配列の最後を超えたアクセス
- for文の中だけを見れば発見できる

```
linux-2.6/drivers/net/wireless/wavelan_cs.c
1700    static int
1701    wv_frequency_list(u_long base,    /* i/o port of the card */
1702                      iw_freq *list,  /* List of frequency to fill */
1703                      int max)        /* Maximum number of frequencies */
1704    {
1705      u_short         table[10];     /* Authorized frequency table */
1706      long            freq = 0L;     /* offset to 2.4 GHz in .5 MHz + 12 MHz */
1707      int             i;             /* index in the table */
1708      const int       BAND_NUM = 10; /* Number of bands */
1709      int             c = 0;         /* Channel number */
...
```

長さ10の配列 “channel_bands” に対し、cによって要素10を参照している

```
1722              while((((channel_bands[c] >> 1) - 24) < freq) &&
1723                     (c < BAND_NUM))
1724                   c++;
```

- 関数wv_frequency_listが決して呼ばれない場合には、バグではない!?

  初期状態から到達可能か否かの判定が必要

  判定する＝モデルチェッキング

  判定しない＝スタティックチェッキング

スタティックチェッキングで発見できる典型的な例

- 普通の意味でバグではあるが、そのままでも正しく動作する可能性大
- 下手にデバッグしない方がよい！？

File: drivers/net/wireless/hostap/hostap_hw.c

If文の後の行へ進む場合は、resは必ず0

```
931  if (res) {
932    printk(KERN_DEBUG "%s: hfa384x_set_rid: CMDCODE_ACCESS_WRITE "
933           "failed (res=%d, rid=%04x, len=%d)¥n",
934           dev->name, res, rid, len);
935    return res;
936  }
937
```

条件”res == -110” は決して真とはならない

```
938           if (res == -ETIMEDOUT)
```

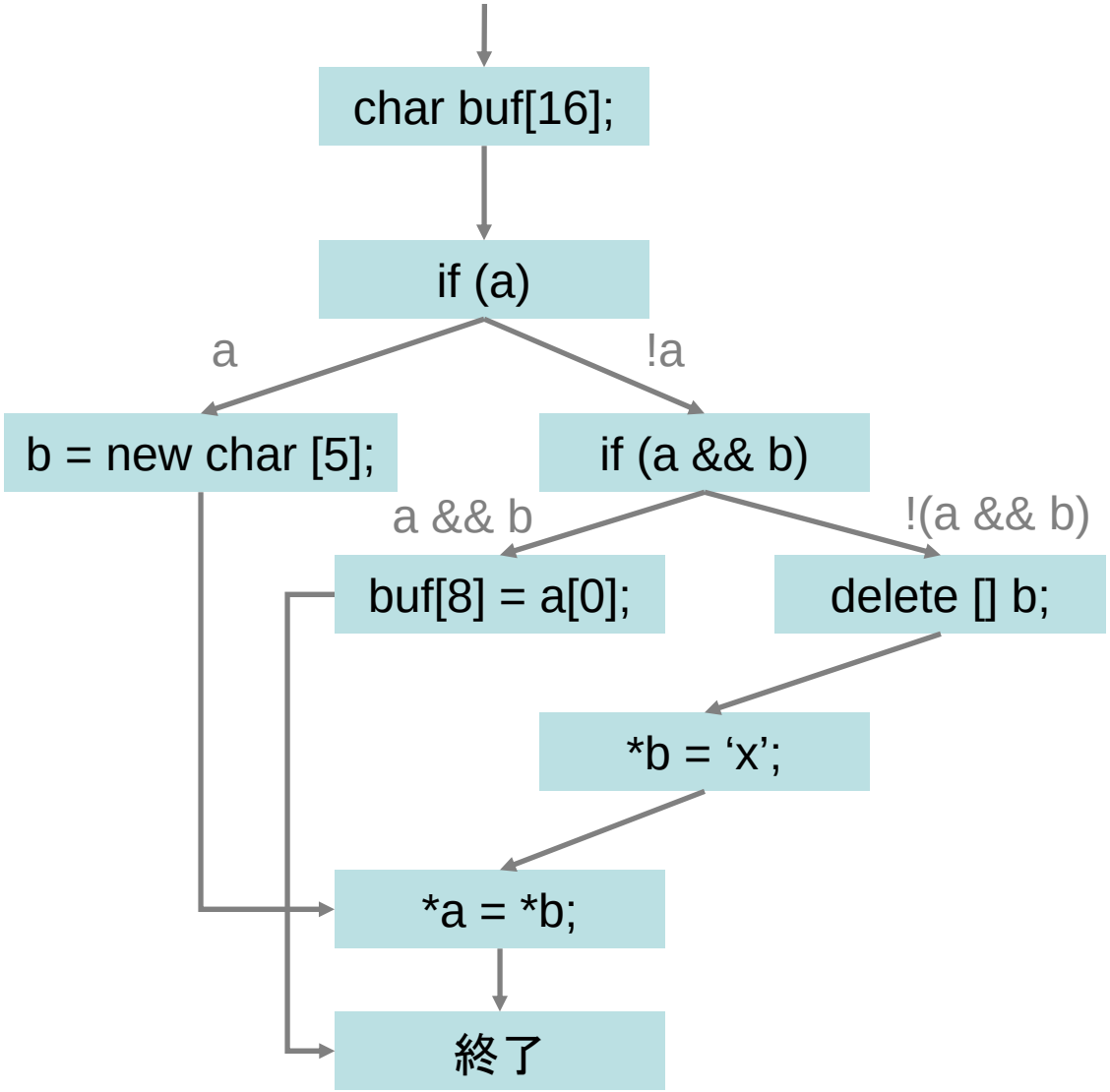
939行が実行されることは決してない

```
939                   prism2_hw_reset(dev);
```

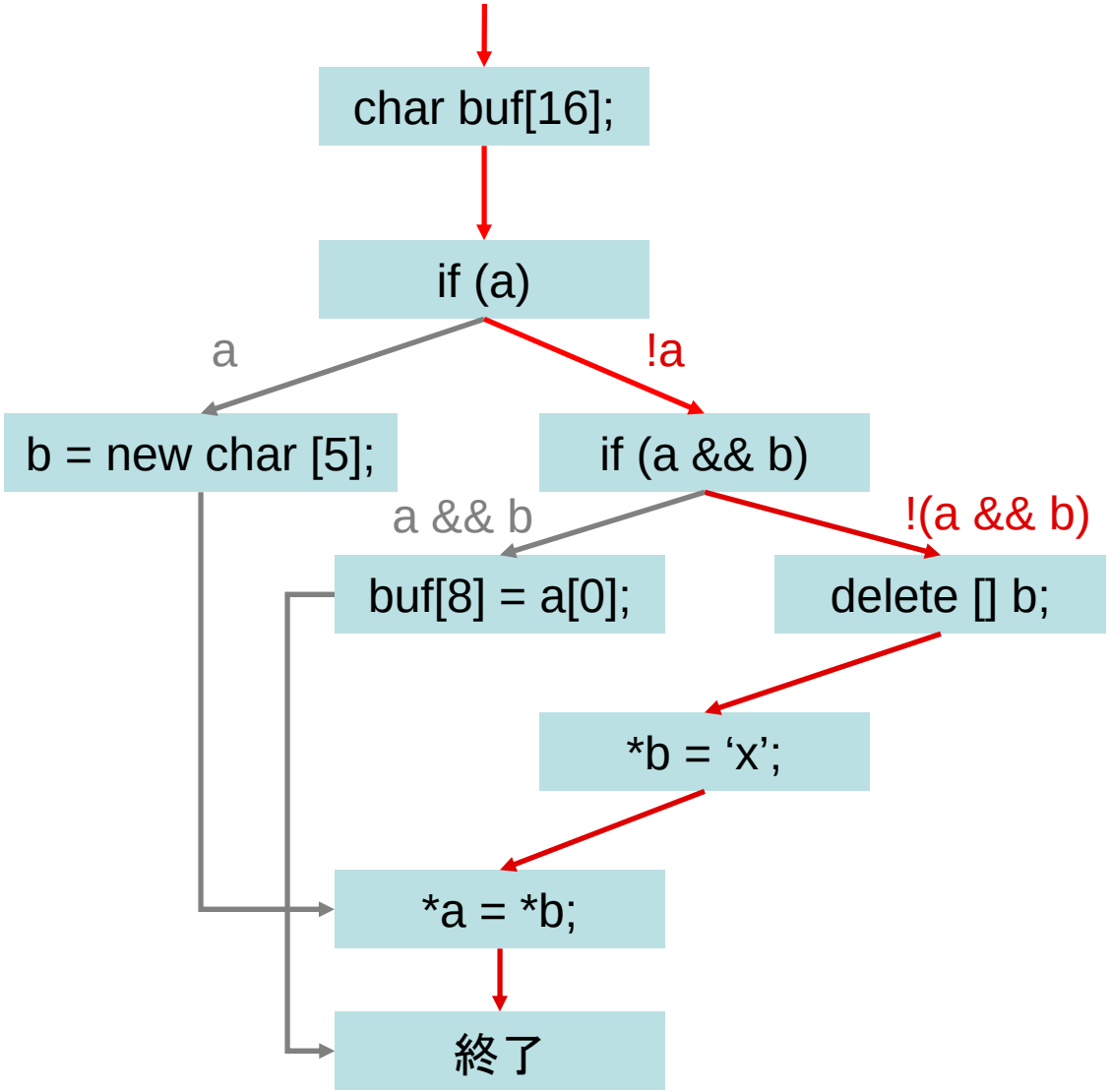
検証例

```
#define SIZE 8
void set_a_b(char * a, char * b) {
    char buf[SIZE];
    if (a) {
        b = new char[5];
    } else {
        if (a && b) {
            buf[SIZE] = a[0];
            return;
        } else {
            delete [] b;
        }
        *b = ‘x’;
    }
    *a = *b;
}
```

コントロールフローグラフ（CDG）を生成



CDG上を探索（ここでは明示的に）

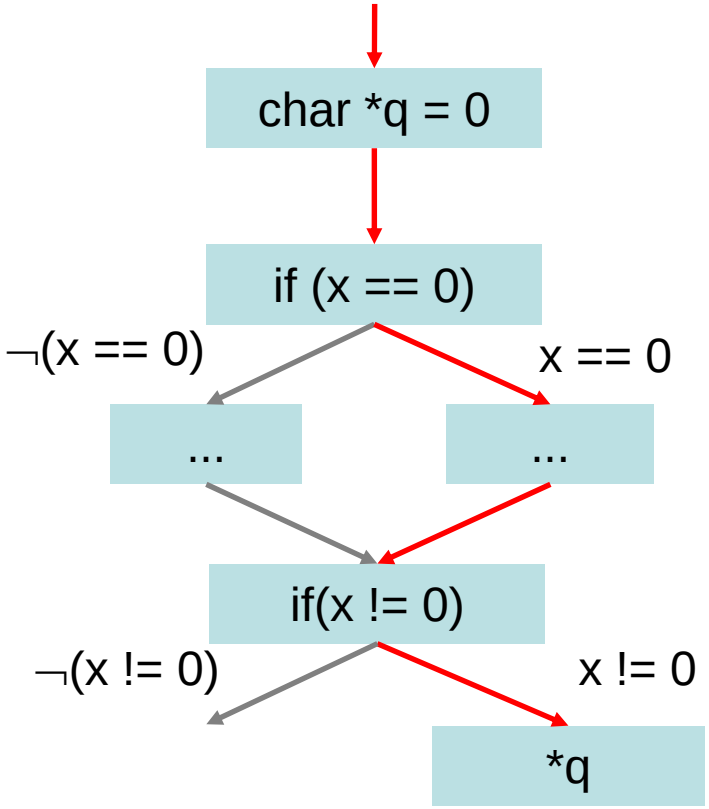


パスを解析（SAT/SMTソルバーを利用）

- データフロー解析を併用
  - ループなど特殊な場合ごとの専用の解析手法を利用
  - 基本はパスごとにデータフロー解析でconservative

実行のfalse path問題

- このパスが実行されることは決してない



Null pointer

p → Null

p → Error

実行パスの条件

$x_0 = 0$

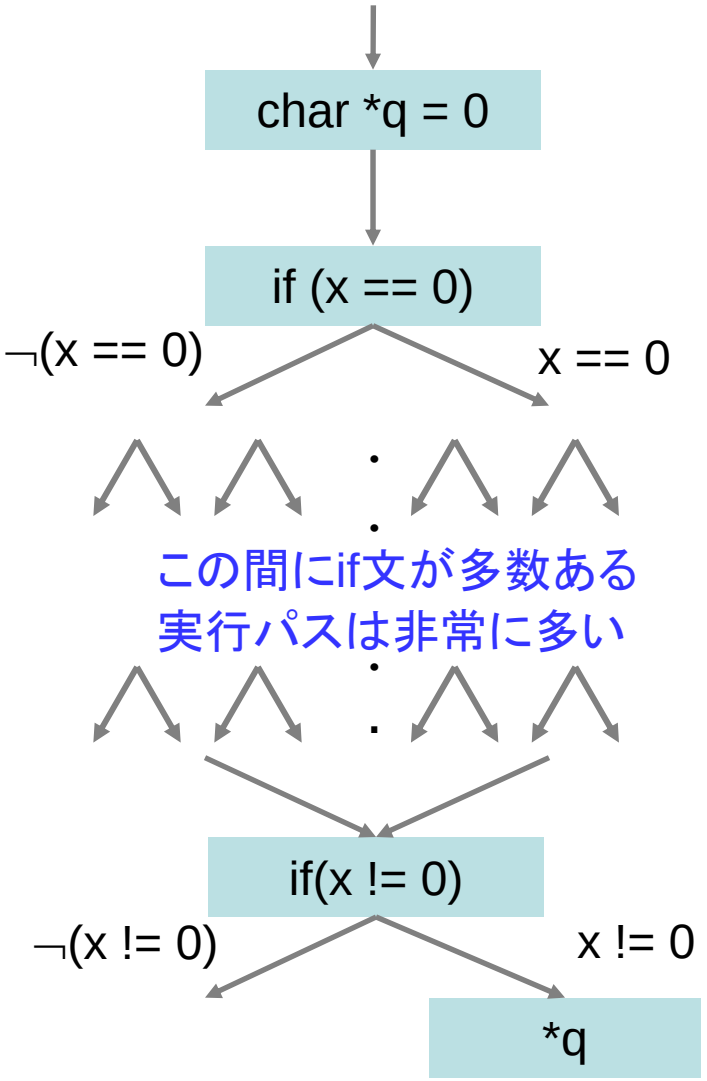
$\wedge \neg(x_0 = 0)$

SAT/SMT
ソルバー

充足不可能

Proof: $\{x_0 = 0, \neg(x_0 = 0)\}$

実際のfalse path解析



指数的に多数のパスが同じ条件により false pathとなる

それらは同じ推論から充足不可能となるだけでなく、条件式の形（シンタックス）も同じになる

一度、SAT/SMTソルバーで解析すれば、多数の実行パスをfalse pathと判定できる

スタティックチェッキングでfalse pathを排除できる場合も多い

SATソルバーにおけるUnsatisfiable coreの計算と基本的に同じ

記号シミュレーション

- 形式的検証のためにC言語記述から、その動作を抽出する基本手法
- C言語記述に対し、最初から順に実行していく
- ただし、0, 2, 101などの固定値ではなく、a, b, cなどの記号値として実行する（シミュレーションする）
- 記号値の範囲に制限をつける場合とつけない場合がある

  整数全体を表現する変数としてのa

  32ビットで表現できる整数としてのa[0:31]

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
            //
            i = -i;
            //
    } else {
            /* 何もしない */
            //
    }
    //
    assert i>0; //
    return i;
}
```

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
        // 実行パス条件: i0<0          i の値は i0
        i = -i;
        //
    } else {
        /* 何もしない */
        //
    }
    //
    assert i>0; //
    return i;
}
```

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
        // 実行パス条件: i0<0          i の値は i0
        i = -i;
        // 実行パス条件: i0<0          i の値は -i0
    } else {
        /* 何もしない */
        //
    }
    //
    assert i>0; //
    return i;
}
```

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
        // 実行パス条件: i0<0          i の値は i0
        i = -i;
        // 実行パス条件: i0<0          i の値は -i0
    } else {
        /* 何もしない */
        // 実行パス条件: !(i0<0)       i の値は i0
    }
    //
    assert i>0; //
    return i;
}
```

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
        // 実行パス条件: i0<0          i の値は i0
        i = -i;
        // 実行パス条件: i0<0          i の値は -i0
    } else {
        /* 何もしない */
        // 実行パス条件: !(i0<0)       i の値は i0
    }
    // 実行パスが融合:  i の値は (i0<0) ? -i0 : i0
    assert i>0; //
    return i;
}
```

記号シミュレーション例（絶対値の計算）

```
int abs(int i) {
    // i0 変数 i の初期値を i0 とする
    if (i < 0) {
        // 実行パス条件: i0<0          i の値は i0
        i = -i;
        // 実行パス条件: i0<0          i の値は -i0
    } else {
        /* 何もしない */
        // 実行パス条件: !(i0<0)       i の値は i0
    }
    // 実行パスが融合:  i の値は (i0<0) ? -i0 : i0
    assert i>0; // 最終的な条件 ((i0<0) ? -i0 : i0 )>0
    return i;
}
```

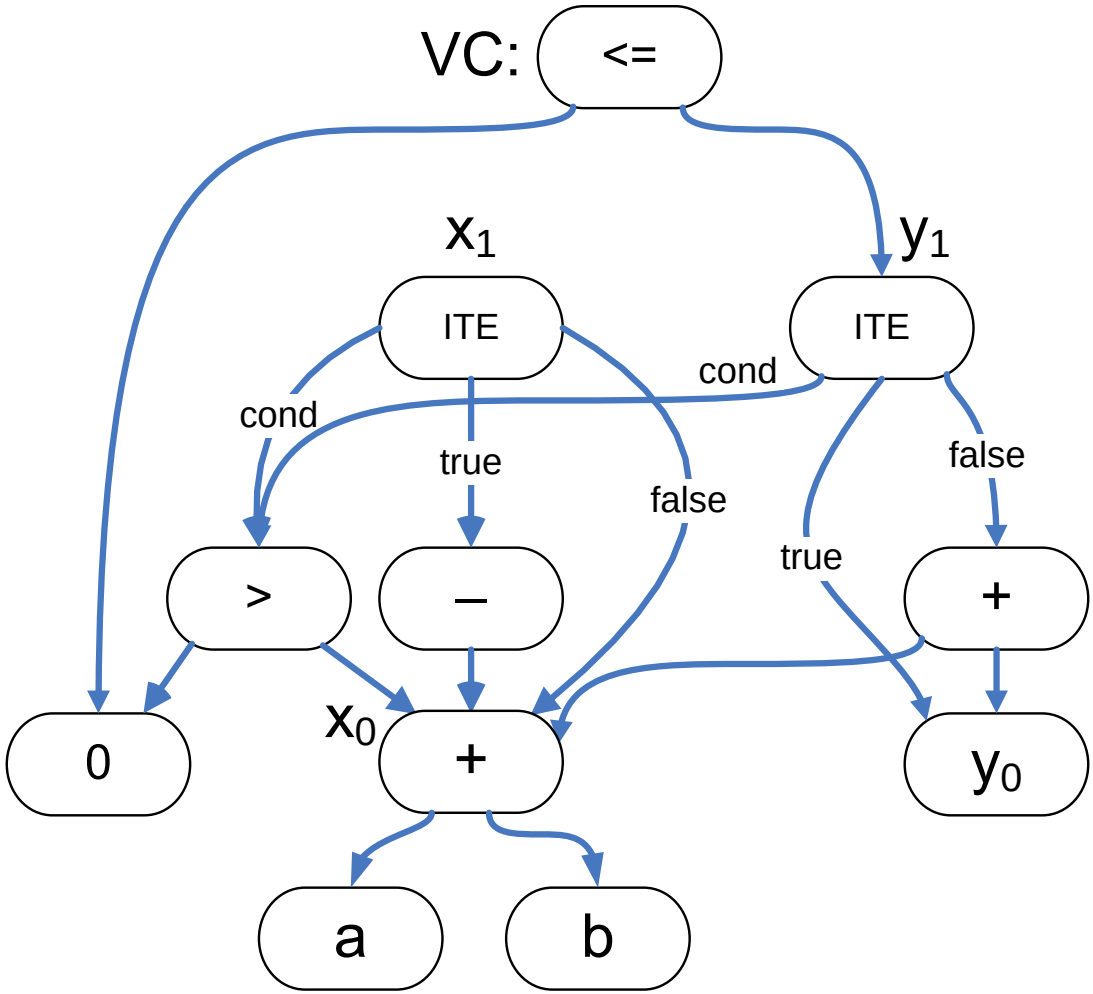
((i0<0) ? -i0 : i0)>0 が成立するか否かをSAT/SMTソルバーなどで調べる

記号シミュレーションの問題点

- 現状のC言語記述に対する形式的検証技術では、もっとも効率的な解析手法
- 問題は、記号式がシミュレーションが進むに連れて、膨大になってしまうという点であった
- しかし、現在では、Maximally shared graphと呼ばれるもので、式を最大限共有して管理することでかなり防げるようになっている
  - ハードウェア設計でリソースを最大限に共有して回路規模を抑える話と基本的に同一

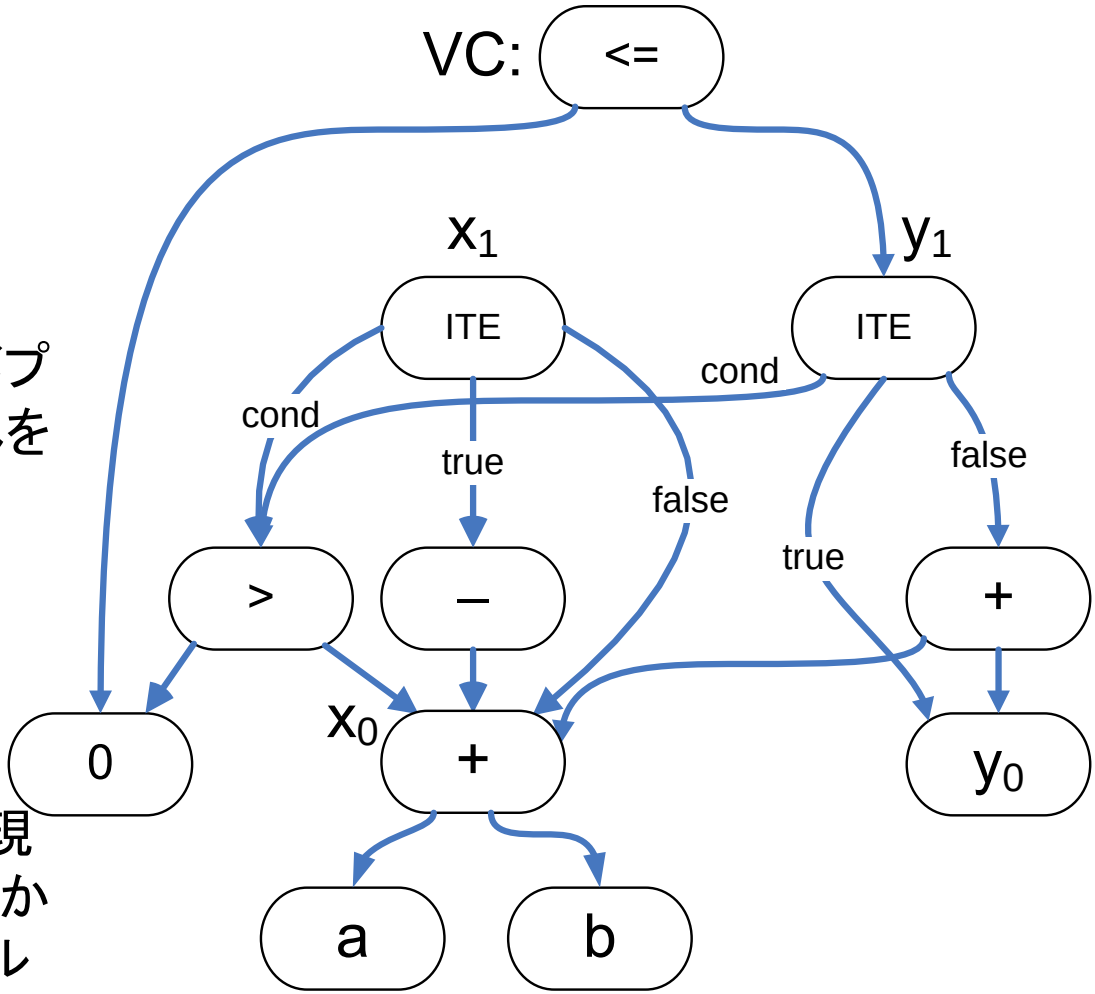
Maximally Shared Graph

x_0=a+b;
if x_0<0 then
    x_1 = -x_0;
    y_1 = y_0;
else
    x_1 = x_0;
    y_1 = y_0+x_0;
assert(0<=y_1);



Maximally Shared Graphによる記号式の管理

- シミュレーション量に比例する大きさ
  - 通常の論理式の表現と等価
- 利点
  - 式の構造が明らか
  - グラフの中のデータの流れがプログラムの中のデータの流れを示す
  - スライシングなど可能
- 構造的冗長性はない
- 正規表現ではない
  - 正規形とするには、新しきが現れる度に、今までの式のどれかと同一か否かをSAT/SMTソルバーでチェックする必要がある



設計の抽象化による大規模プログラムの検証: Predicate abstraction

- プログラム中に現れる変数が数が多すぎる
  - 各変数が代入されるたびに新しい変数が必要
- 何らかの方法で少数の predicatesを選ぶ
  - それぞれの predicate はプログラムの状態や変数の値による、true か false が決まる
  - 従って、predicate の集合により、任意のプログラムの状態をpredicate の数だけの要素を持つビット列（bit vector）の変換できる
- 生成されたビット列間の遷移に対して、検証を実施する
  - False negativeの可能性

Predicate abstraction 手法

プログラムの状態

Predicates

x>0　x>y

抽象状態

x= -1; y=1

00

x= 23; y=11

01

x= -100; y= -1

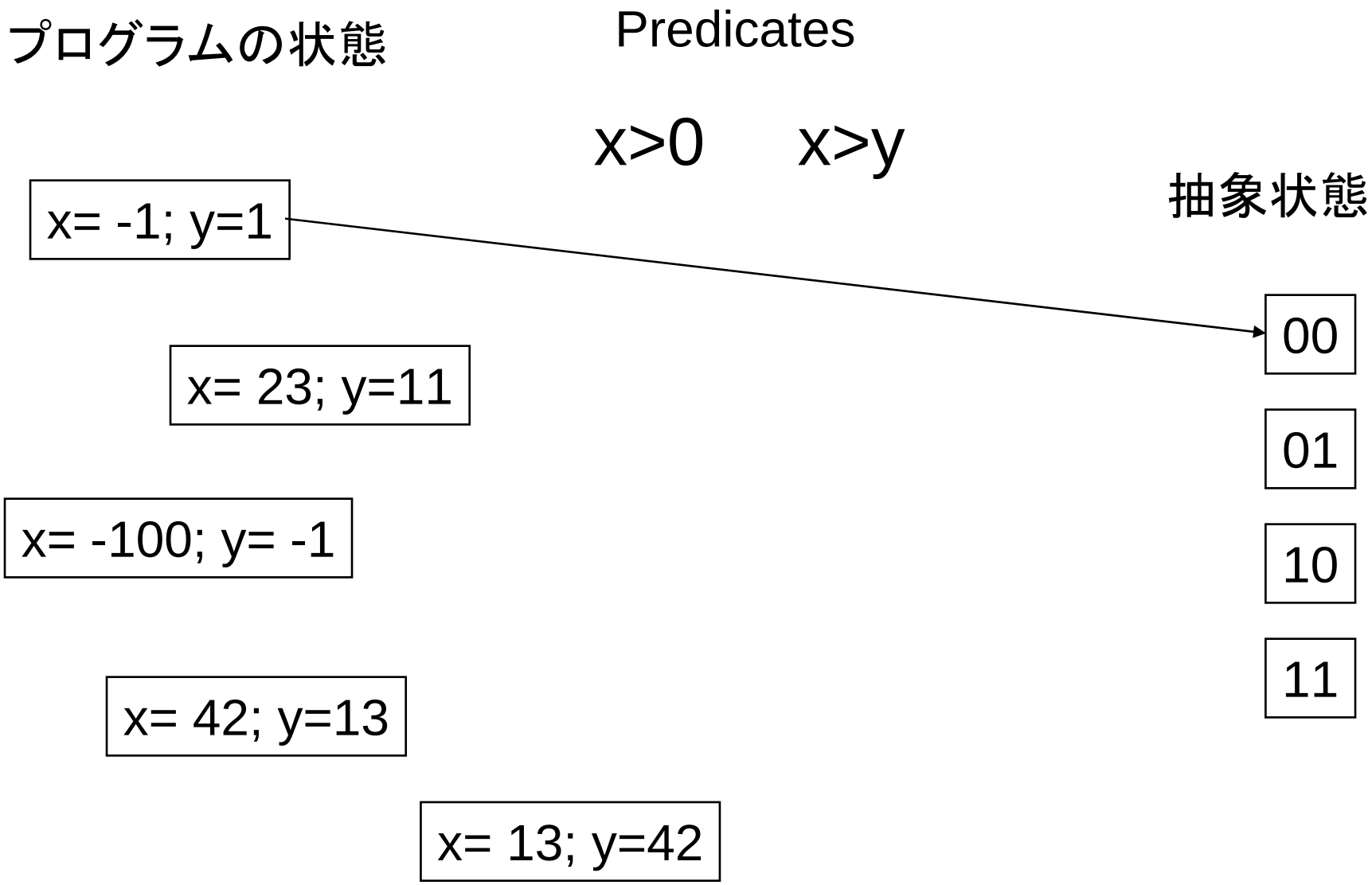
10

x= 42; y=13

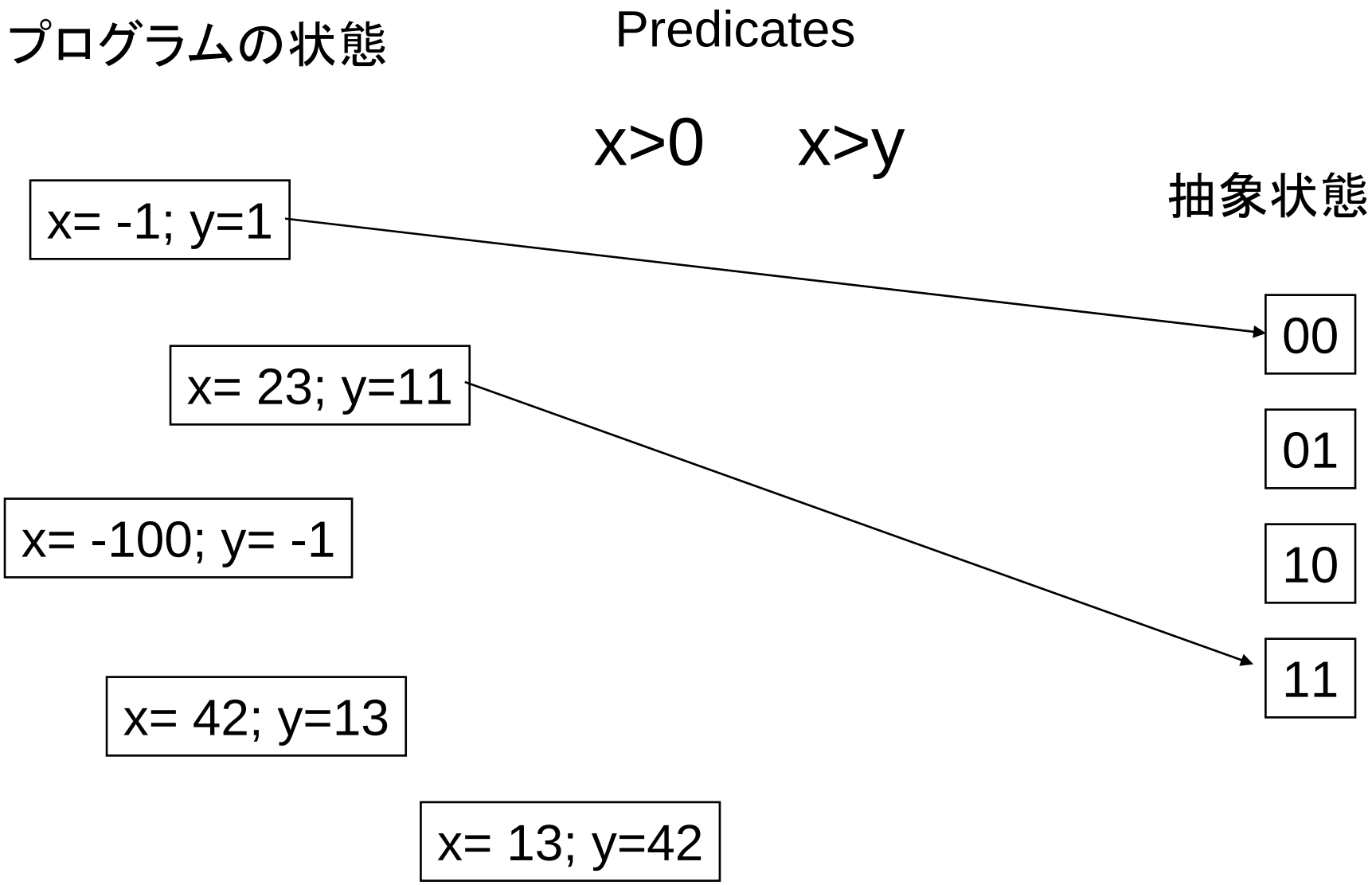
11

x= 13; y=42

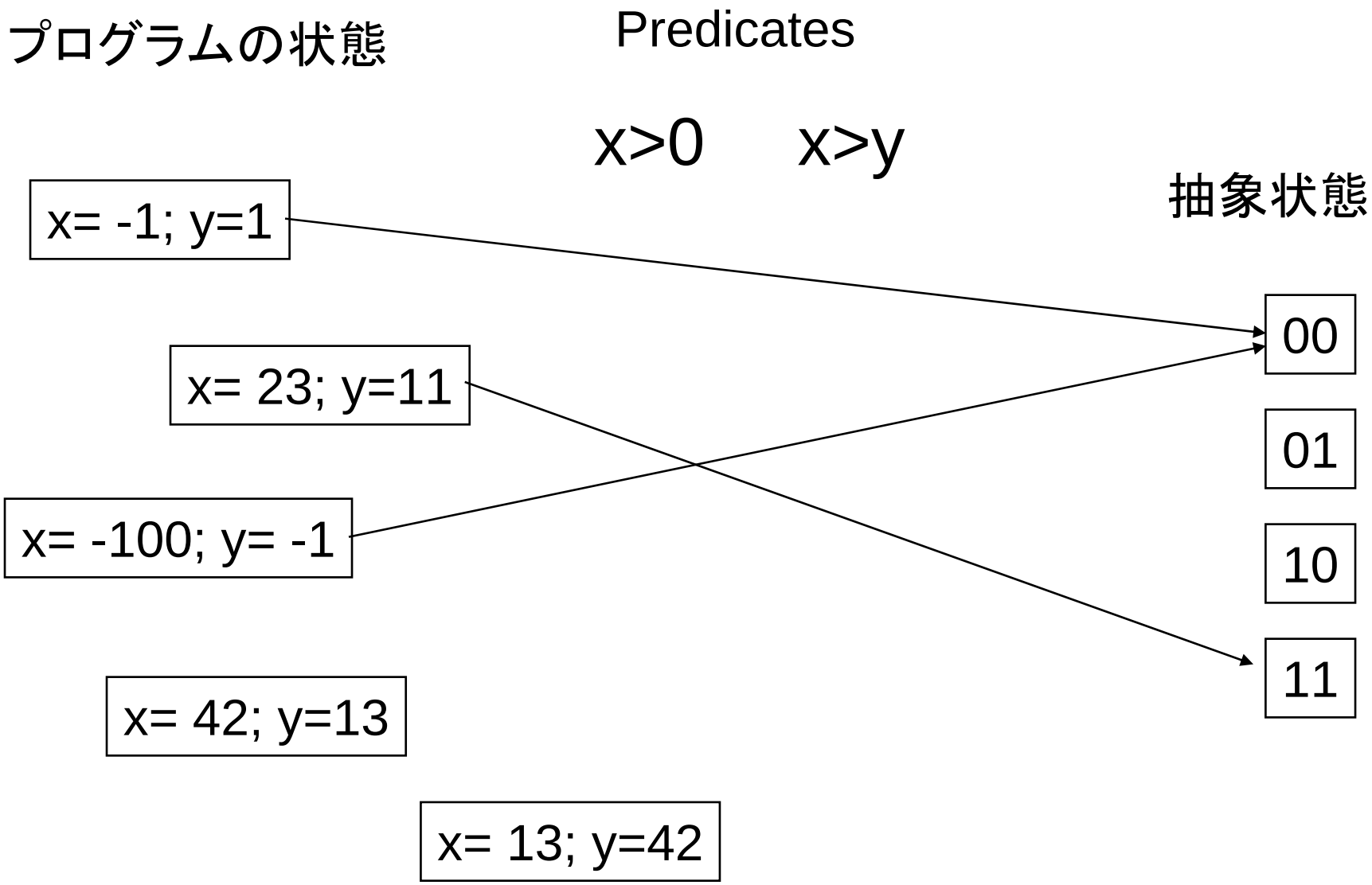
Predicate abstraction 手法



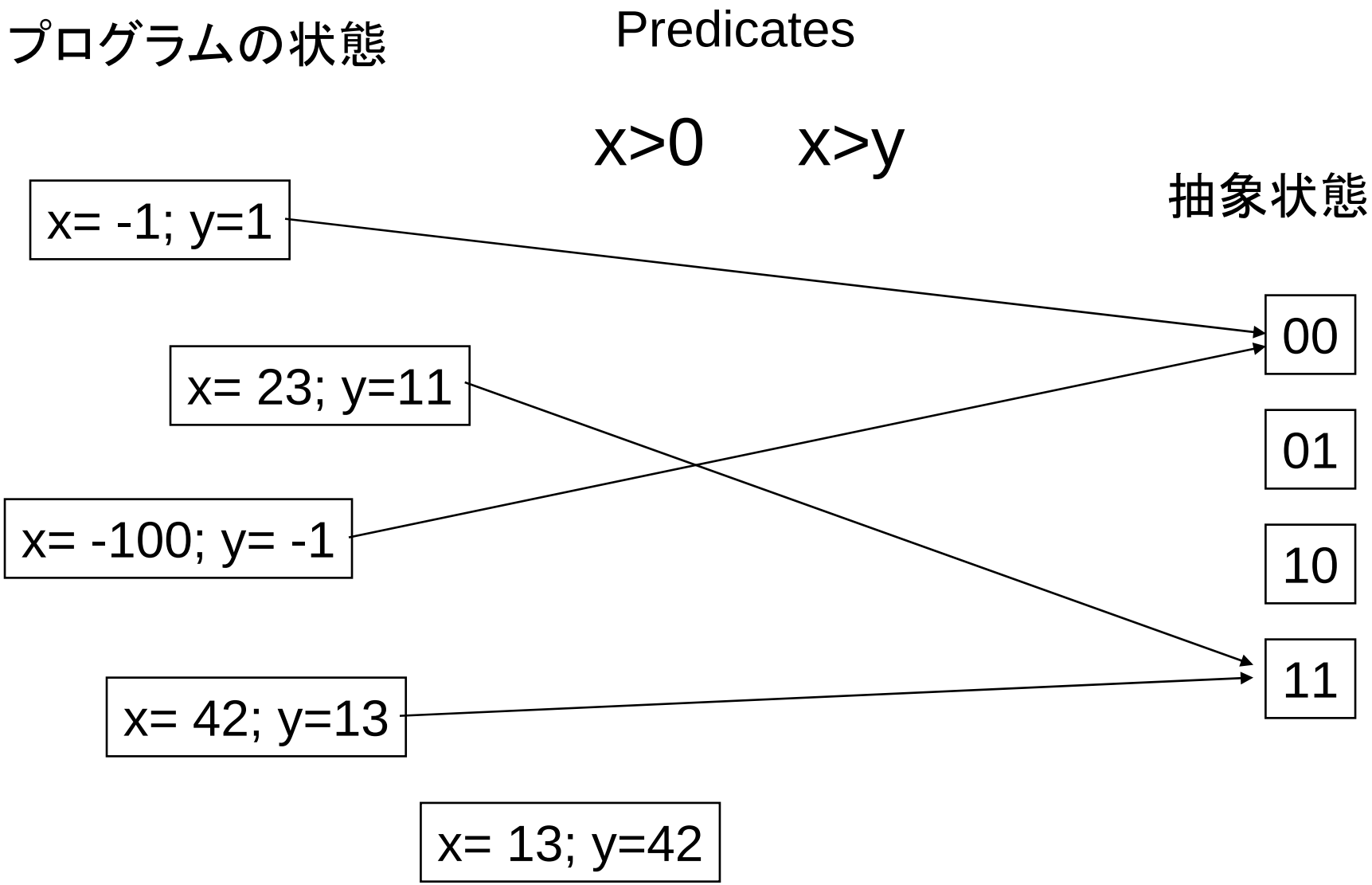
Predicate abstraction 手法



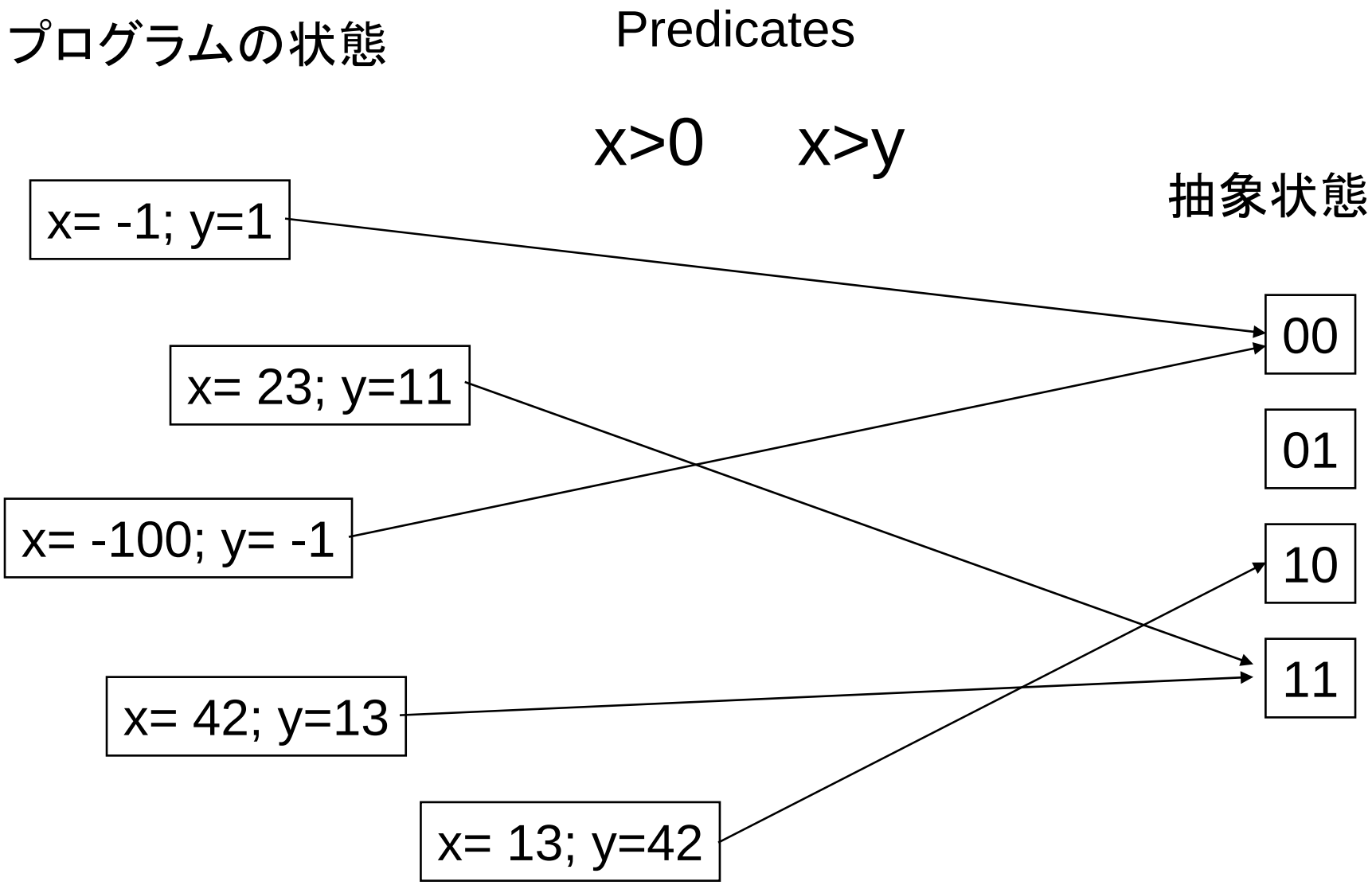
Predicate abstraction 手法



Predicate abstraction 手法



Predicate abstraction 手法



C言語記述を抽象状態空間へ変換

- C言語記述は *Boolean program*と呼ばれる 各predicateに対するBoolean 変数 のみを利用するものに変換可能
  - 変換は conservativeに行う:
    - 元のC言語記述において、ある変数のある値により遷移可能な場合には、Boolean program 上でも必ず遷移可能にする
  - False negativeの可能性
    - CEGAR（Counter Example Guided Abstraction Refinement）も活発に研究されている
  - Boolean program は non-deterministic

変換例

- 次の2つのpredicateを利用する
  - p: x>0
  - q: x>y
- 文 x++; は次のようになる:

```
if (p && q) { /* 何もしない, p も q も true のまま */}
else if (p && !q) { q = non_deterministic(0,1); }
else if (!p && q) { p = non_deterministic(0,1); }
else {
    p = non_deterministic(0,1);
    q = non_deterministic(0,1);
}
```

predicate p: i>0 の場合のabs() の例

```
int abs(int i) {
    //
    if (i < 0) {
        //
        i = -i;
        //
    } else {
        /* 何もしない */
        //
    }
    //
    assert i>0; //
    return i;
}
```

predicate p: i>0 の場合のabs() の例

```
int abs(int i) {
    //
     if (!p && non_det(0,1)) {
            //
            i = -i;
            //
    } else {
            /* 何もしない */
            //
    }
    //
    assert i>0; //
    return i;
}
```

predicate p: i>0 の場合のabs() の例

```
int abs(int i) {
    //
     if (!p && non_det(0,1)) {
          //
          p = p ? 0 : non_det(0,1);
          //
    } else {
          /* 何もしない */
          //
    }
    //
    assert i>0; //
    return i;
}
```

predicate p: i>0 の場合のabs() の例

```
int abs(int i) {
    //
     if (!p && non_det(0,1)) {
          //
          p = p ? 0 : non_det(0,1);
          //
    } else {
          /* 何もしない */
          //
    }
    //
    assert p; //
    return i;
}
```

Predicate abstractionの特徴

- 利点
  - 大規模記述が扱える
  - 検証で正しいとされた記述は正しい
- 欠点
  - False negativeが多数でる場合が多い
  - Predicate の選択が極めて重要
    - 人手で行うは、そのプログラムの中身をよく知っていないと難しいし、また間違えやすい
    - 自動化手法は種々提案されているが、不十分
- ツール例: Slam, Blast, SATAbs
  - 制御系のソフトウェアの検証ではうまくいっている (デバイスドライバの2重ロック検出など)
- CEGAR（Counter Example Guided Absractin Refinement）の利用

進歩が著しい分野

- CBMC (Clarke, Kroening, Yorav, 2003 CMU)
  - ANSI C記述を検証できる初めての検証ツール
  - 数千行規模の記述まで
- Calysto (Babic, Hu, 2007 UBC)
  - ビットレベルの記号シミュレーションに基づく
  - スタティックチェッキングも融合
  - 記号シミュレーション方の改良
  - Abstraction/refinement の導入
  - 新しいSMTソルバー
- Z3ベースの検証ツール（Microsoft）
  - SMTソルバーであるZ3の性能を活かした検証ツール、シミュレーションベクター自動生成ツール、など多くの研究開発が進んでいる

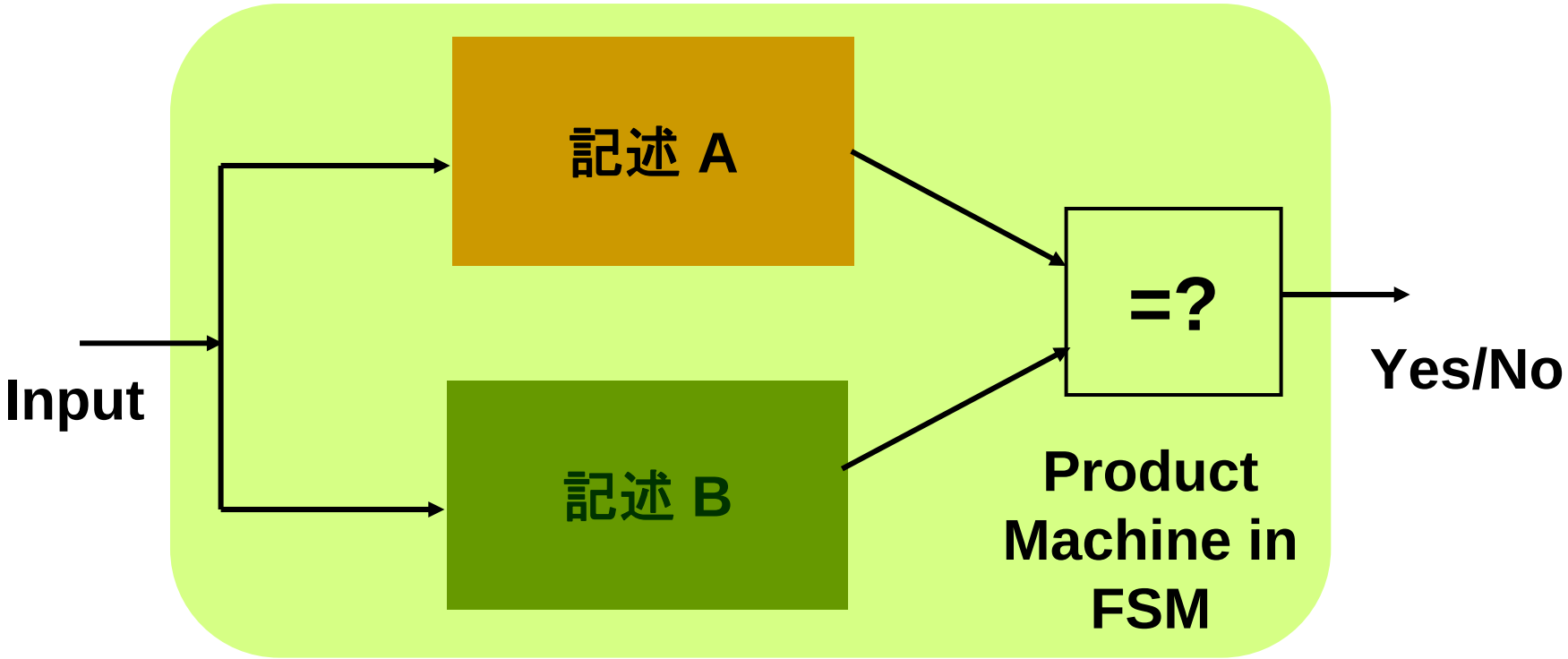
Calysto の性能

Benchmark	LOC	Reports	Bugs	Unkn.	FP Rate	Time [s]
Bftpd 1.8	4532	12	11	0	9%	3.14
Bftpd 1.9.2	4602	5	4	0	20%	2.86
HyperSAT 1.7	9123	0	0	0	0%	14.57
Spin 4.3.0	28394	0	0	0	0%	6858.10
Openssh 4.6p1	81908	4	1	0	75%	8995.64
Inn 2.4.3	122727	10	6	1	34%	1312.33
Ntp 4.2.4p2	185865	30	26	0	14%	558.16
Ntp 4.2.5p66	192019	13	4	3	56%	493.39
Openldap 2.4.4a	374266	20	15	2	27%	200.02
Bind 9.4.1p1	393318	5	2	3	0%	*2436.88
TOTAL	1406754	99	69	9	23%	20875.09

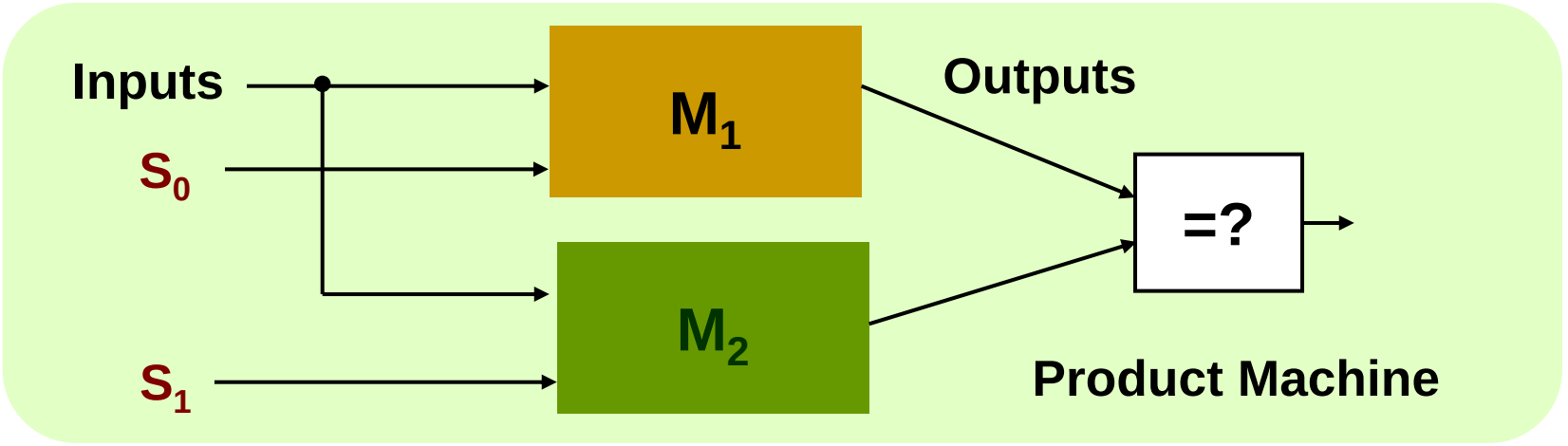
http://www.domagoj-babic.com/ に最新データ

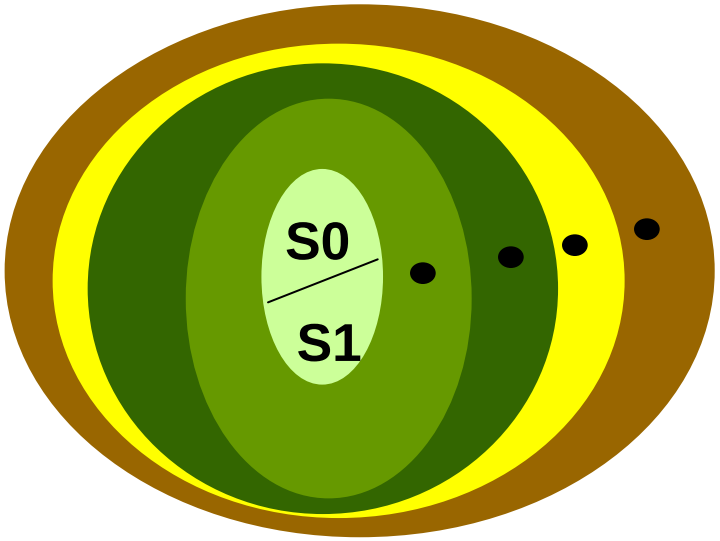
形式的等価性検証

与えられた2つの記述に対し、可能な全ての入力シーケンスに対し、対応する出力が等価であるか否かを調べる



状態空間の探索による等価性検証

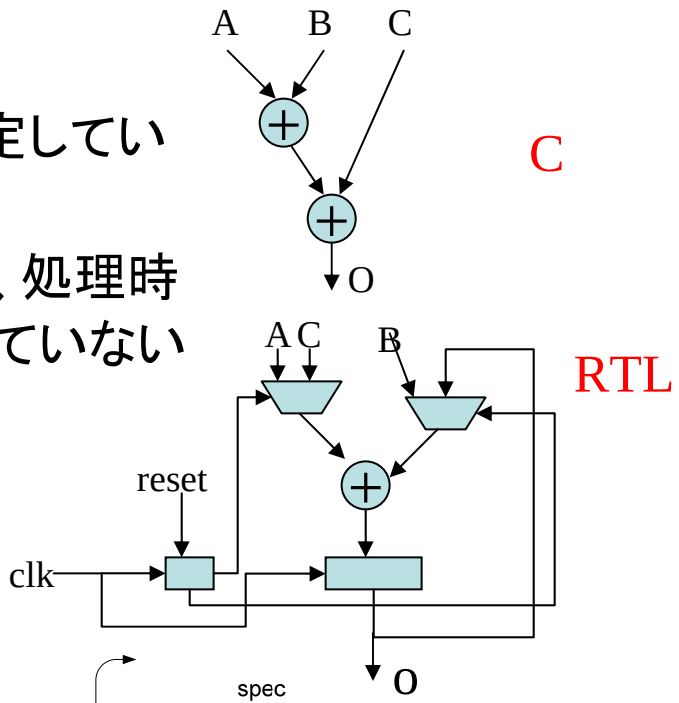


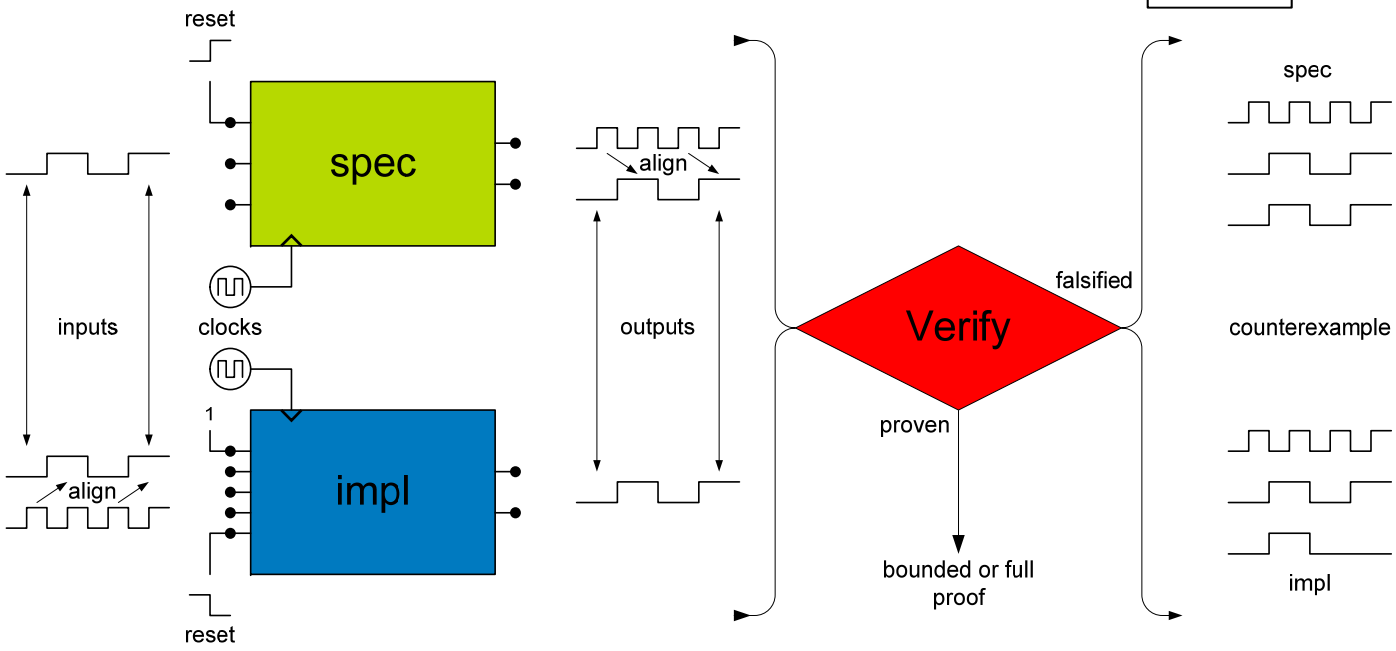


- of M_1 と M_2 から product machine を作る
- リセット状態 S_0, S_1から順にproduct machineの状態遷移を辿っていく
- 指定された状態に対して、対応する出力値が等価か調べる
- 任意の状態空間探索手法が使える

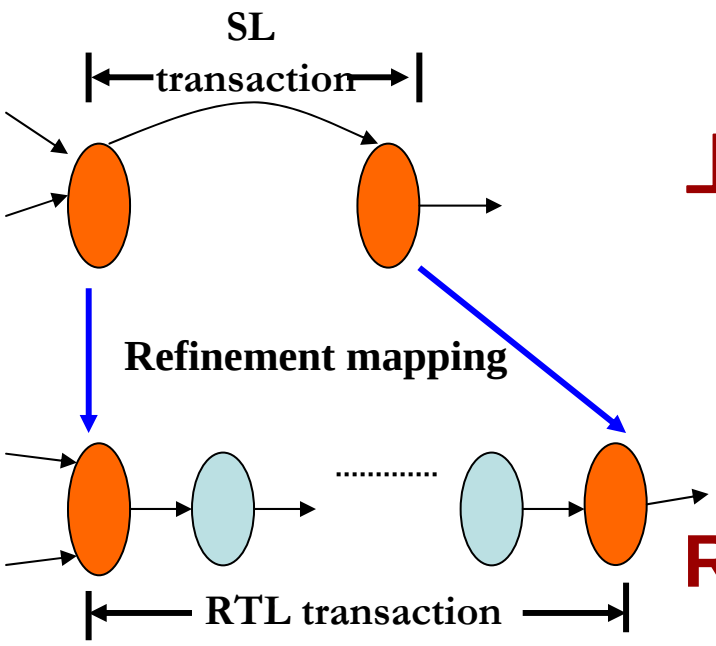
C言語ハードウェア設計における等価性検証問題

- スケジューリングの違い
  - RTLでは、各クロックサイクルの動作が決定している
  - C言語レベルでは、順番は決まっているが、処理時刻は部分的にしか決っていない／全く決っていない
  - 組合せ回路の等価性検証とは異なる
  - 初期状態・リセット状態の対応
  - 入力と出力のタイミングの対応





トランザクション



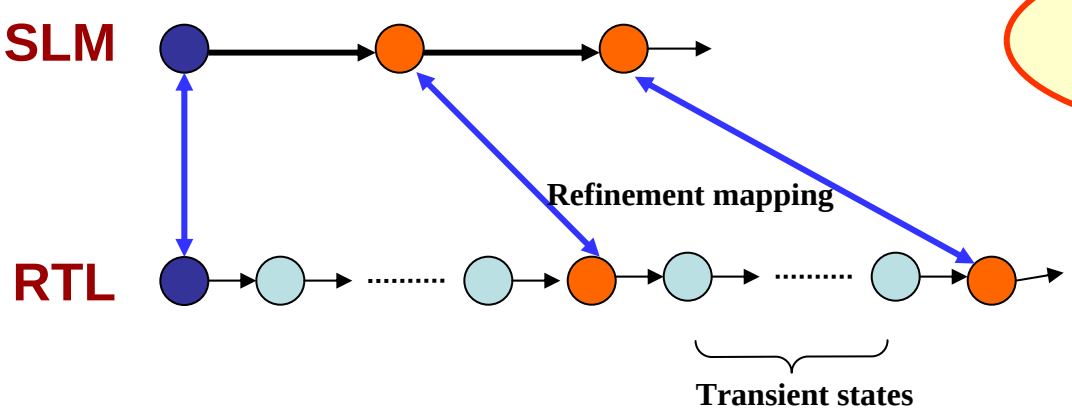
上位（システムレベル）

RTL

- 上位設計における計算の単位
- 実行の終了時には、同期する状態に到達する
- 組合せ回路に対する等価性検証における、内部等価点（potential equivalence points）と 同じアイデア

トランザクションレベルの等価性検証

- RTLの状態の内、上位レベル（システムレベル）に対応する状態があるものを「同期する状態」と呼ぶ
- 同期する状態ごとに検証問題を分割できる
  - 基本は記号シミュレーションによる等価性検証
  - 一般に、完全な検証には帰納法を利用する必要がある
- 組合せ回路に対する等価性検証における、内部等価点（potential equivalence points）と 同じアイデア



Complete Verification

or

Sequential counterexample

C言語記述に対する等価性検証の基本

- 記号シミュレーションの利用
  - 実行パスを順に探索していく
    - 大きな記述では多数の実行パスがあり、記号シミュレーションが困難になる
- Equivalence Class (EqvClass)の抽出
  - 等価と判定された式／変数などを集めたもの
  - 記号シミュレーション中に生成される
    - 代入文の左辺と右辺は等価
    - 簡単な式変形で等価と判明するもの
- 最終的には、SAT/SMTソルバーで等価性を判断
  - どこまで検証できるかは、SAT/SMTソルバーの性能による
  - ループなどに対しては、専用の解析手法を利用

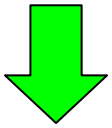
簡単な例

```
a = v1;
b = v2;
add1 = a + b;
```

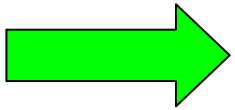
記述 1

```
add2 = v1 + v2;
```

記述 2



E1 (a, v1)
E2 (b, v2)
E3 (add1, a+b)
E4 (add2, v1+v2)



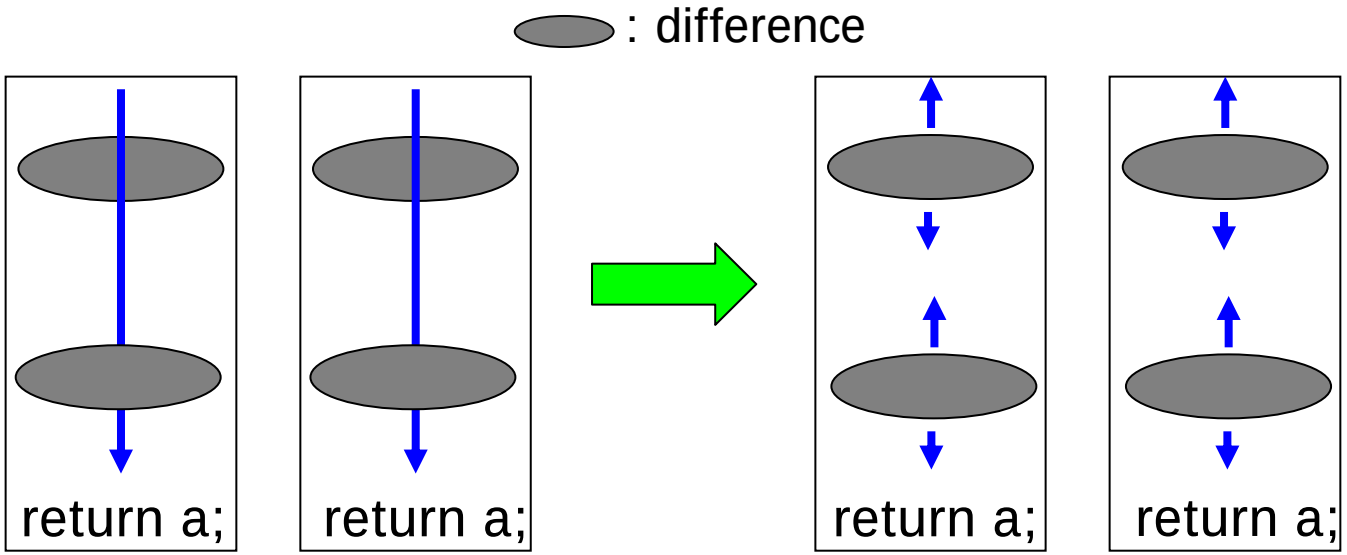
E1 (a, v1)
E2 (b, v2)
E3’ (add1, a+b,
add2, v1+v2)

4つの代入文から4つの EqvClasses が生成される

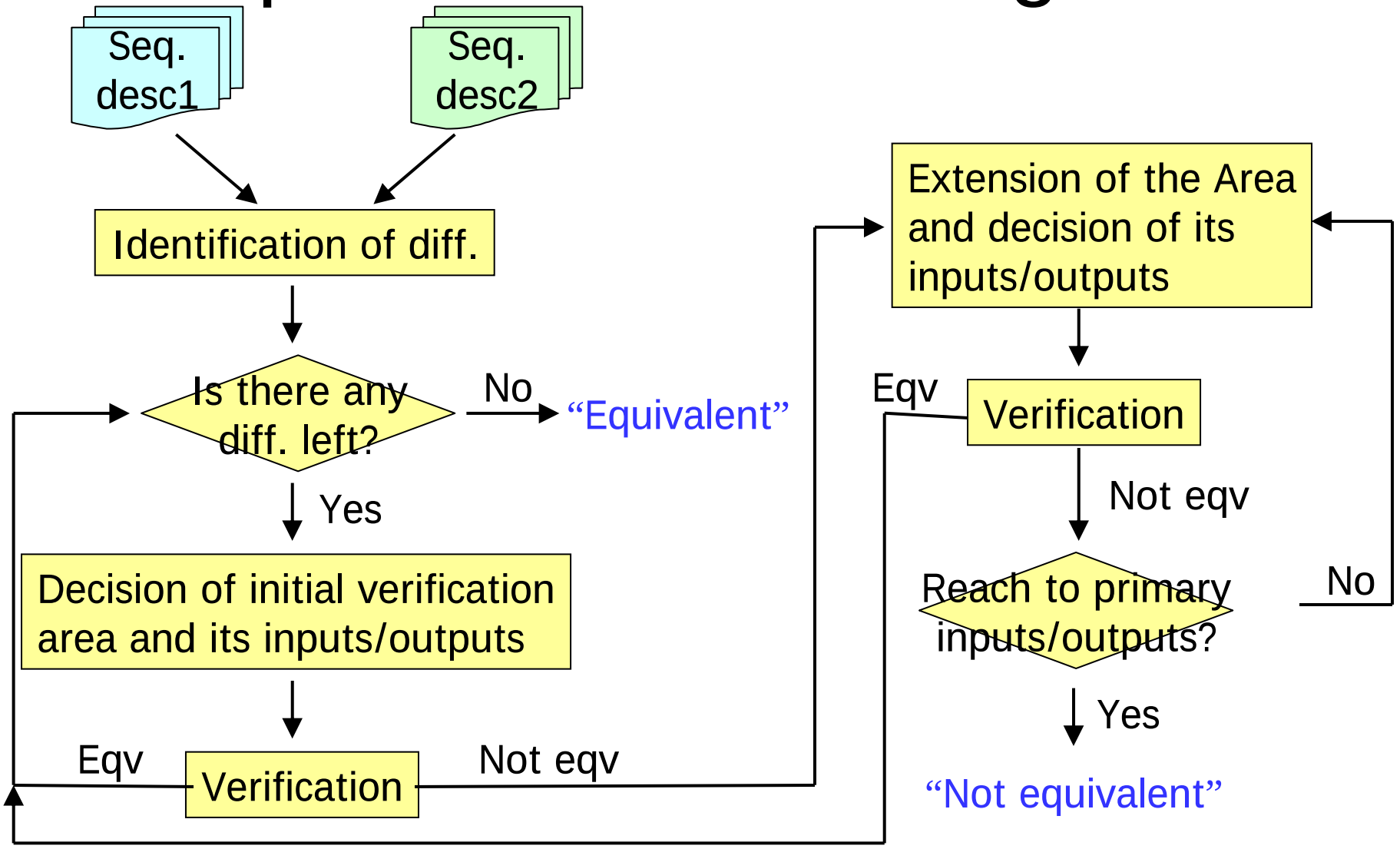
a と *v1* が等価、*b* と *v2* が等価であるため、
E3 と E4 は同一になる

問題点と我々のアプローチ

- 大規模記述に記号シミュレーションは適用できない
  - 実行パスが膨大
    - パス条件をマージすると条件式が膨大
  - 我々のアプローチ: 記述間の差異に注目し、記号シミュレーションしなければならない範囲をできるだけ限定する



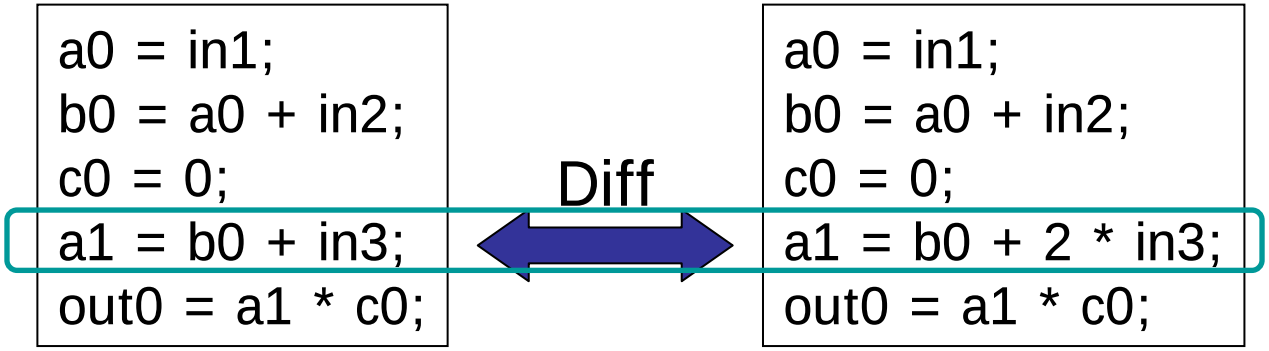
Equivalence Checking Flow



検証領域とその入出力

- 検証領域：差異に基づく文の集合
- 入力変数
  - 検証領域で利用される変数
- 出力変数
  - 検証領域で代入され、検証領域外で利用される変数
- 検証問題
  - 同じ名前の出力変数は等価であるか？
  - 対応する入力変数の等価性を利用

例

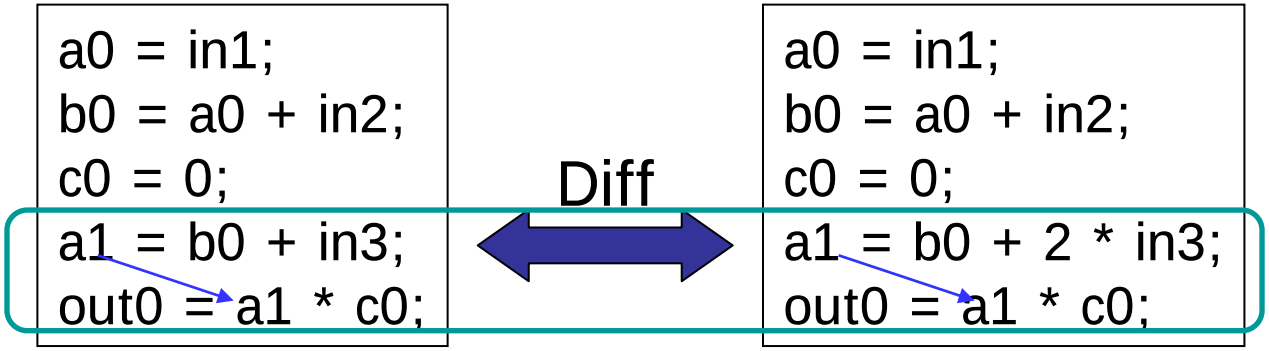


(※) in1, in2, in3 は外部入力とする

- 第1回検証
  - 入力 … b0, in3
  - 出力 … a1
  - 検証結果 … 等価性を証明できず

a1 から出力側へ領域を拡大する

例

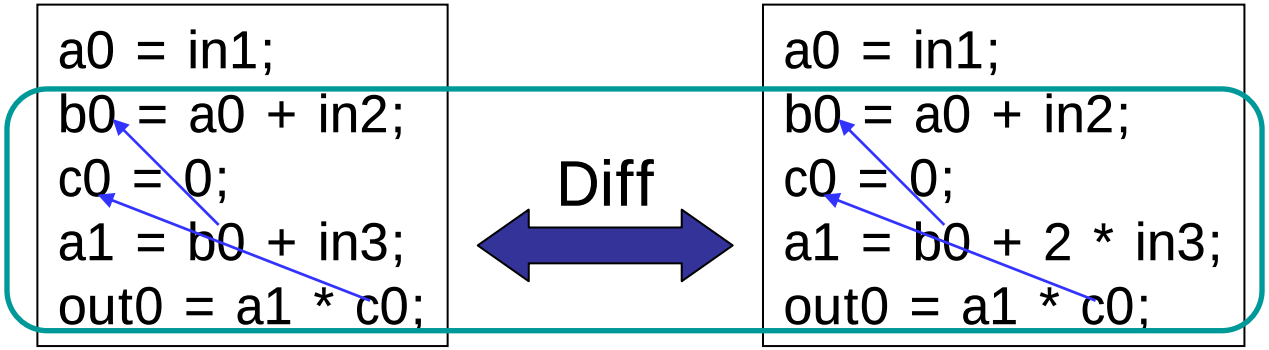


(※) in1, in2, in3 は外部入力とする

- 第2回検証
  - 入力 … b0, c0, in3
  - 出力 … out0
  - 検証結果 …等価性を証明できず

入力側へ検証領域を拡大する

例



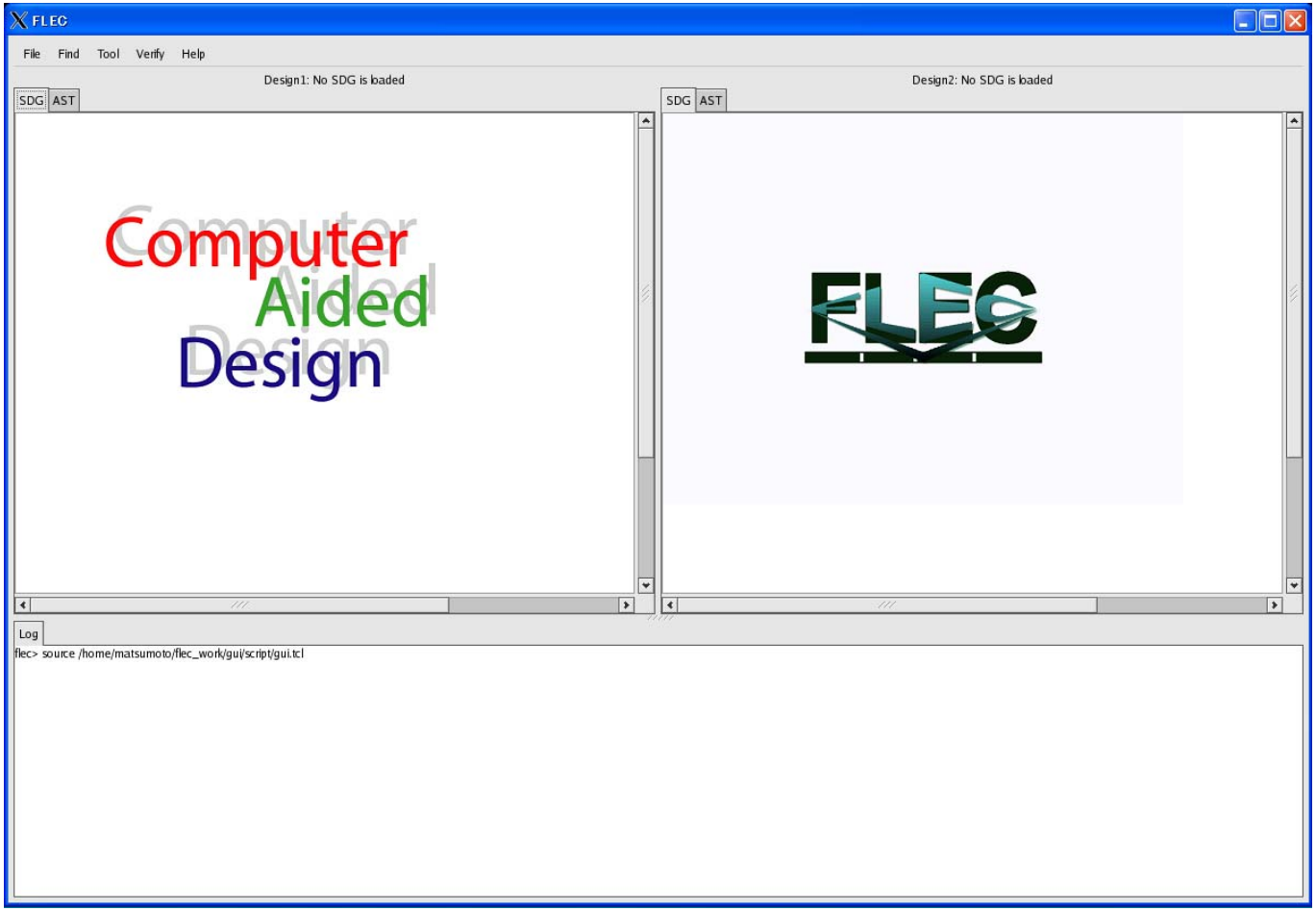
(※) in1, in2, in3 は外部入力とする

- 第3回検証
  - 入力 … a0, in2, c0 (= 0), in3
  - 出力 … out0
  - 検証結果 … 等価（c0 = 0なので）

差異部分は等価ではなかったが、検証領域を拡大することで等価と検証された

等価性検証環境：FLEC

- FLEC (Fujita Lab Equivalence Checker)
  - Funded by STARC (2003-2007) and CREST (2007~)



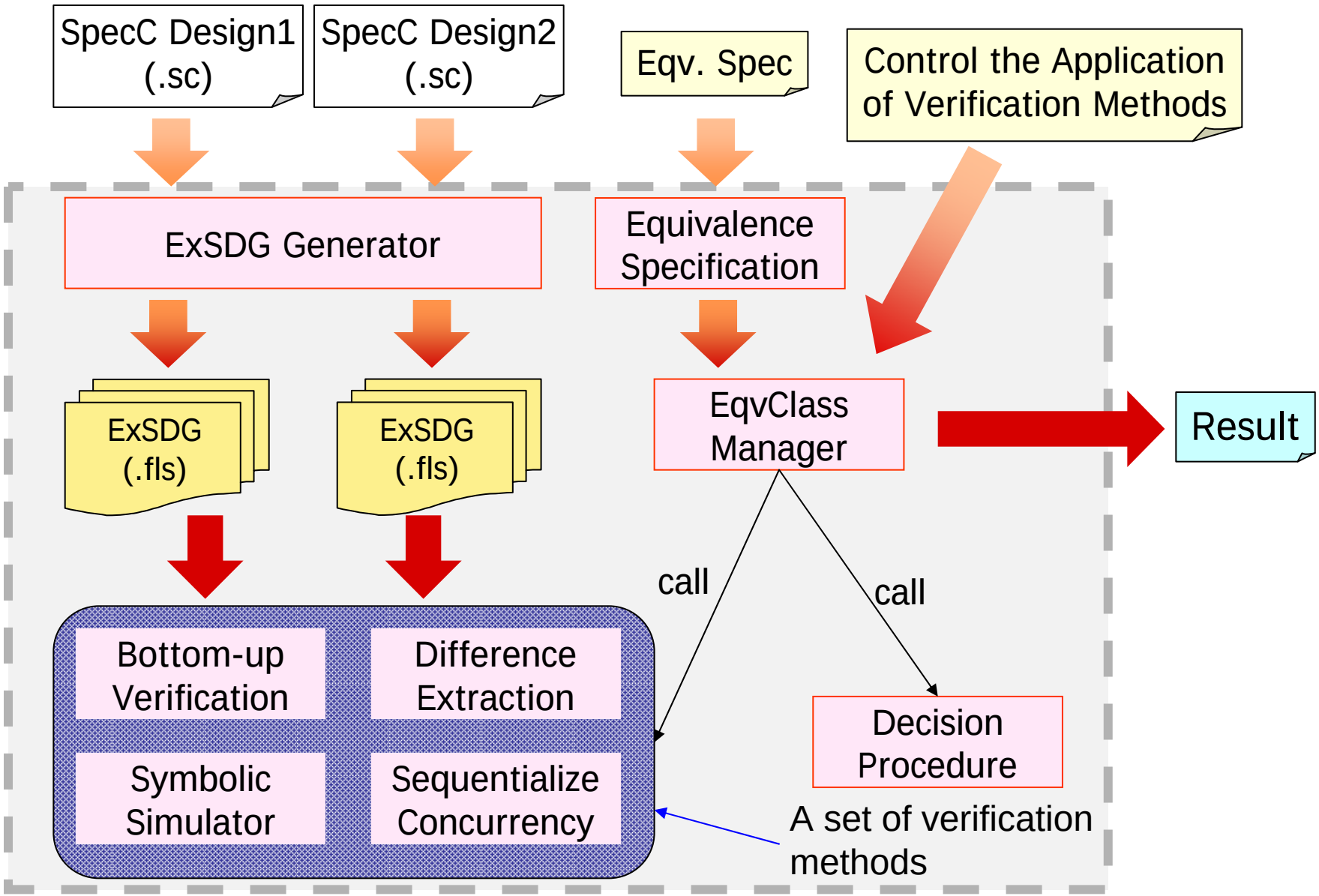
FLEC の特徴

- STARCとの5年間の共同研究、CRESTの元での研究（5年の予定）
- 形式的な解析手法の組合せ
  - いくつかの手法が実装され、記述に応じて切り替える
- 依存グラフに基づく実装
  - 与えられた設計記述を一旦、ExSDG (Extended System Dependence Graph)に変換
    - SDG と AST を統合したデータ構造
  - ExSDGを順次解析することで検証
    - 必要に応じて抽象化などにより、異なるデータ構造へも変換する
- 実行時間付き記述の取り扱い可能
  - “latency” と “throughput”による等価性をユーザが定義

扱える記述

- Untimed and Timed Behavior Level Design
  - 並列実行 (par in SpecC)
  - 同期 (wait/notify in SpecC)
  - 時間制御 (waitfor in SpecC)
- RTL Design
  - Cycle accurate な動作
  - Wire 変数をサポート (SpecC の signal)
- 制限
  - Pointer, recursive call, dynamic memory allocation

ツールの構成


SpecC Design1 (.sc)
SpecC Design2 (.sc)
Eqv. Spec
Control the Application of Verification Methods
ExSDG Generator
Equivalence Specification
ExSDG (.fls)
ExSDG (.fls)
EqvClass Manager
Result
call
call
Bottom-up Verification
Difference Extraction
Symbolic Simulator
Sequentialize Concurrency
Decision Procedure
A set of verification methods

取り扱うSpecC記述

- ExSDG は SpecC IR (SIR)から生成
  - SystemCからのトランスレータも計画中

```
behavior Adder1(in int in1,
  in int in2, in int in3,
  out int out1) {
  void main() {
    out1 = in1 + in2 + in3;
  }
};
```

Untimed Behavior Level

```
behavior Adder2(in int in1,
  in int in2, in int in3,
  out int out1) {
  void main() {
    int tmp;
    while(1) {
      tmp = in1 + in2;
      waitfor(5);
      tmp = tmp + in3;
      waitfor(5);
      out1 = tmp; }
  }
};
```

Timed Behavior Level

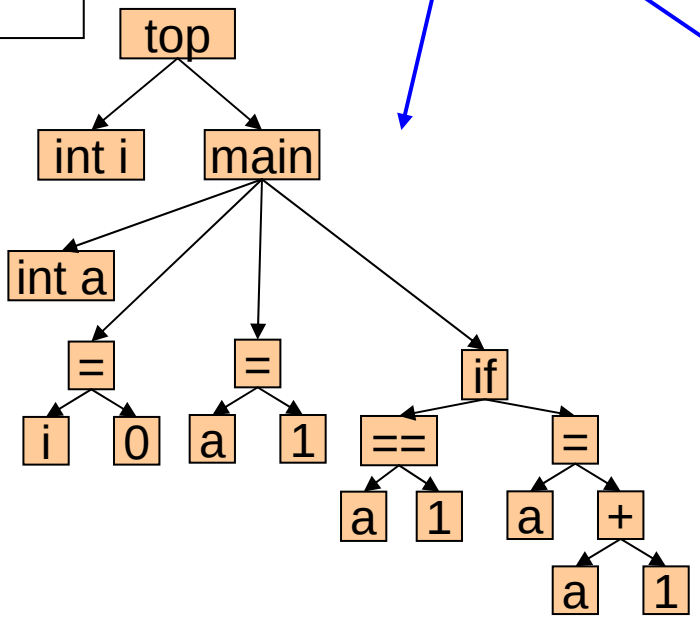
```
behavior Adder3(in int in1,
  out int out1) {
  void main() {
    int tmp;
    while(1) {
      tmp = in1; waitfor(1);
      tmp = tmp + in1; waitfor(1);
      tmp = tmp + in1; waitfor(1);
      out1 = tmp; waitfor(1);
    }
  }
};
```

Register Transfer Level

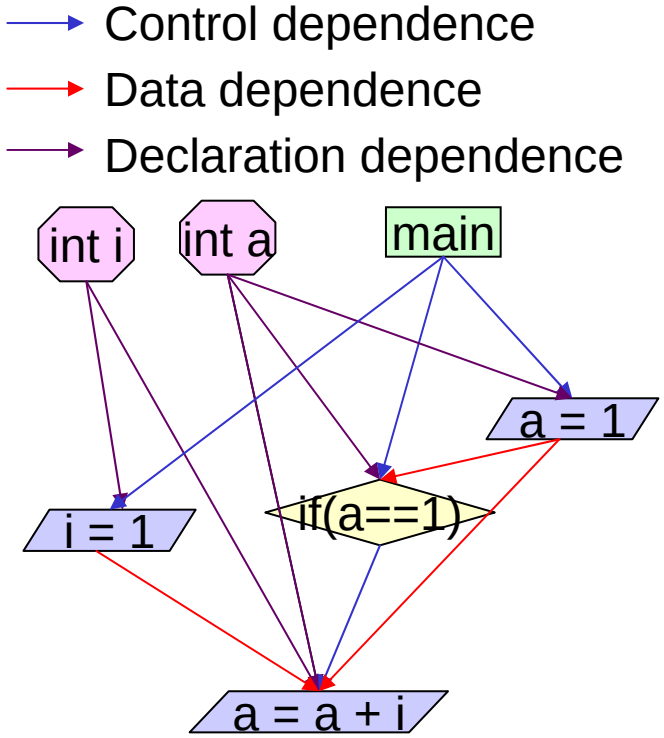
ExSDG (Extended System Dependence Graph)

```
int i;
main(){
    int a;
    i = 1;
    a = 1;
    if(a == 1)
        a = a + i;
}
```

ノードの対応が不明確
⇒　使いにくい



Abstract Syntax Tree (AST)

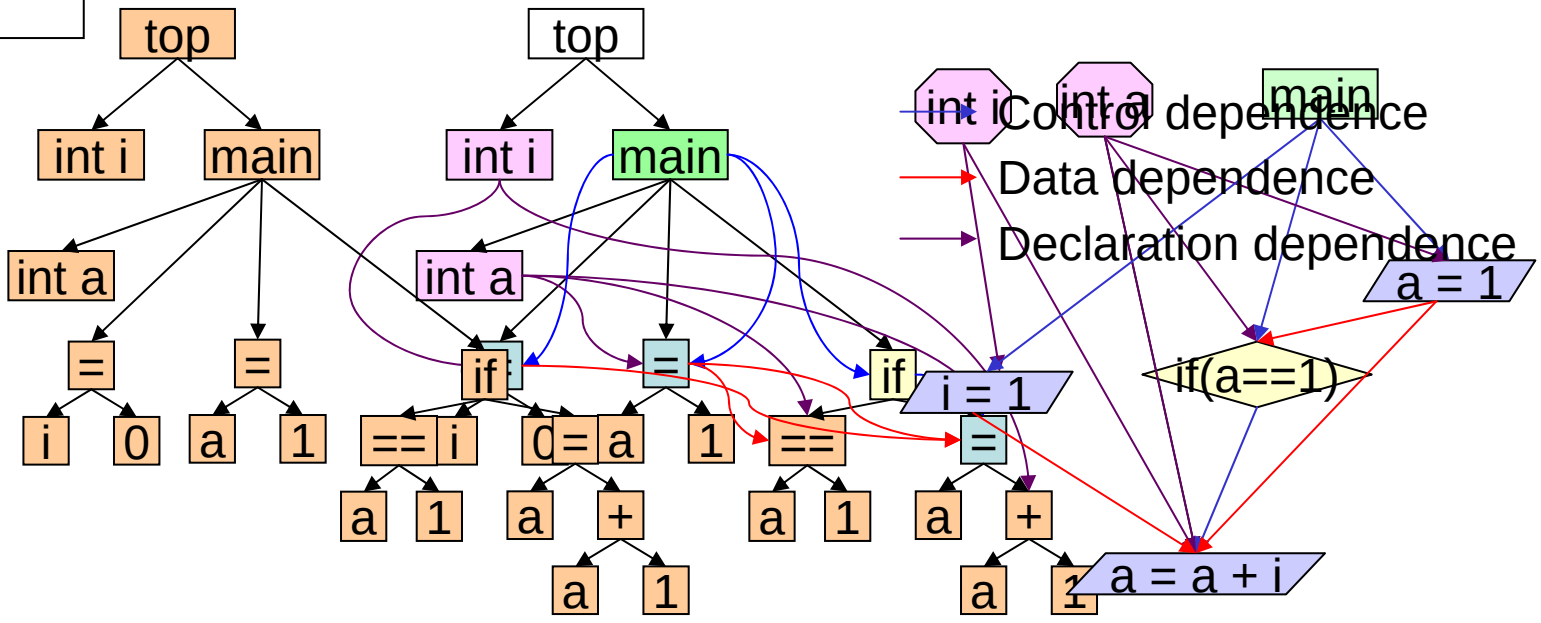


System Dependence Graph (SDG)

ExSDG (Extended System Dependence Graph)

```
int i;
main(){
    int a;
    i = 1;
    a = 1;
    if(a == 1)
        a = a + i;
}
```

- ExSDG により:
  - 各種解析・形式的検証ツールを容易に実装可能
  - システムレベル、トランザクションレベル、RTL設計を表現
  - C / C++ / SpecC / SystemC / SystemVerilog / Verilog-HDL / VHDLから変換可能
    （現在はSpecCからのトランスレータのみ実装）



Abstract Syntax Tree (AST) **Extended System Dependence Graph (ExSDG)** **System Dependence Graph (SDG)**

等価性指定

- 等価性は次の4つから指定する
  - (Port, Throughput, Latency, Condition)

```
behavior Adder1(in int in1,
  in int in2, in int in3,
  out int out1) {
  void main() {
    out1 = in1 + in2 + in3;
  }
};
```

(in1, 1, 0, TRUE)
(in2, 1, 0, TRUE)
(in3, 1, 0, TRUE)
(out1, 1, 0, TRUE)

```
behavior Adder2(in int in1,
  in int in2, in int in3,
  out int out1) {
  void main() {
    int tmp;
    while(1) {
      tmp = in1 + in2;
      waitfor(5);
      tmp = tmp + in3;
      waitfor(5);
      out1 = tmp; }
  }
};
```

(in1, 10, 0, TRUE)
(in2, 10, 0, TRUE)
(in3, 10, 5, TRUE)
(out1, 10, 10, TRUE)

```
behavior Adder3(in int in1,
  out int out1) {
  void main() {
    int tmp;
    while(1) {
      tmp = in1; waitfor(1);
      tmp = tmp + in1; waitfor(1);
      tmp = tmp + in1; waitfor(1);
      out1 = tmp; waitfor(1);
    }
  }
};
```

(in1, 4, 0, TRUE)
(in1, 4, 1, TRUE)
(in1, 4, 2, TRUE)
(out1, 4, 3, TRUE)

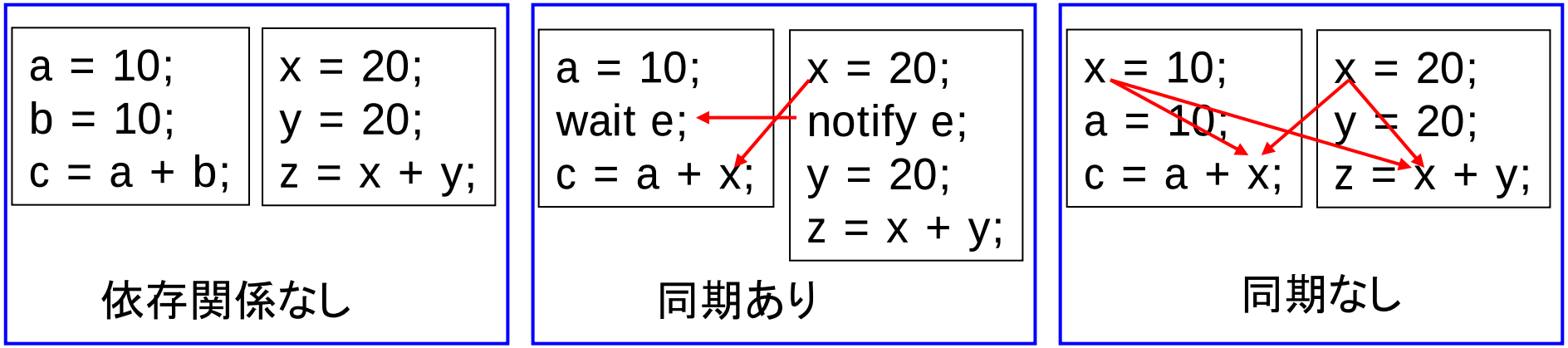
並列記述の扱い

- 並列記述は最初に順序化する
- 文 st1 と文 st2 が並列実行されており、“write-write” あるいは “read-write” の関係にある場合, 次を調べる:
  - P1: always T(st1) > T(st2)
  - P2: always T(st1) < T(st2)
  - T(s) … 文 s の実行時刻
- ILP で定式化して検証
  - P1 (P2) を満たす実行例は存在するか？
  - SpecCにおけるタイミング制御文から条件を生成する
  - 一種の抽象化に基づくモデルチェッキング

(P1, P2) = (pass, pass)	Impossible
(P1, P2) = (fail, pass)	Always st1→st2
(P1, P2) = (pass, fail)	Always st2→st1
(P1, P2) = (fail, fail)	Order is undecidable

順序化可能とする

並列記述の順序化



No check is needed

```
a = 10;
b = 10;
c = a + b;
x = 20;
y = 20;
z = x + y;
```

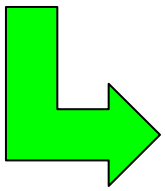
always x=20 → c=a+x?

Result: YES

always x=a+x → x=20?

Result: NO

順序化可能！



```
a = 10;
x = 20;
y = 20;
z = x + y;
c = a + x;
```

always x=10 → x=20?

Result: NO

always x=20 → x=10?

Result: NO

順序化不可能！

ケーススタディ 1: MPEG4

- 2つの記述の差
  - IDCT内の定数伝播、共通式抽出など
- 記述量
  - SpecCで約6,300行
  - ExSDGは約50,000ノード、36,000エッジ
    - ExSDG 生成時間: 780 秒

		Nodes in diff	Result	Run time	# of ext.
MPEG4_org	MPEG4_rev1	96	Eqv	3.3 sec	0
MPEG4_org	MPEG4_rev2	96	Not eqv	13.2 sec	80

ケーススタディ 2: Elevator Controller

- 2つの記述の差
  - 制御系コードの最適化
- 記述量
  - SpecCで約3,300行
  - ExSDGは約20,000ノード、20,000エッジ
    - ExSDG 生成時間: 178 秒

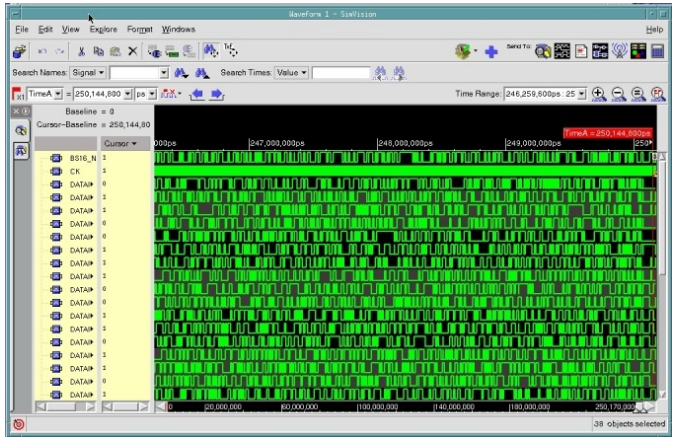
		Nodes in diff	Result	Run time	# of ext.
Elv_org	Elv_rev1	4	Eqv	1.8 sec	1
Elv_org	Elv_rev2	3	----	> 12 hours	4

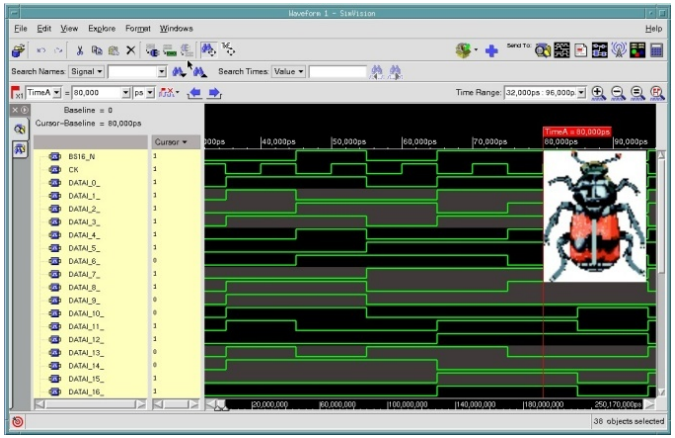
まとめ

- シミュレーションだけでは検証しきれないという共通認識
  - ハードウェア設計者1人に対して、検証エンジニア2人
- SAT/SMTソルバーの研究が進んでいる
  - 効率的な記号シミュレーションと組合せて、数千行規模の検証が可能
- 大規模記述の検証（モデルチェッキング、等価性検証）
  - 記述の抽象化（抽象化の自動改良）
  - 記述の性質を利用した検証（記述間の差異、ループ処理用解析手法）
  - よく考えて利用できれば、ある程度実用的
- スタティックチェッキング
  - 数百万行で扱え、また全自動
  - False warningの問題
    - 数が多いとだれも見ない。バグではあるが直そうとすると返ってバグを増やしてしまう
- デバッグ支援が必須になりつつある
  - シミュレーション／エミュレーションによる長い反例が出たが、何のことかさっぱりわからない（反例を縮小化する）
  - 形式的解析手法を応用したデバッグ支援ツールも登場
    - DAC2009でのデバッグに関するspecial session

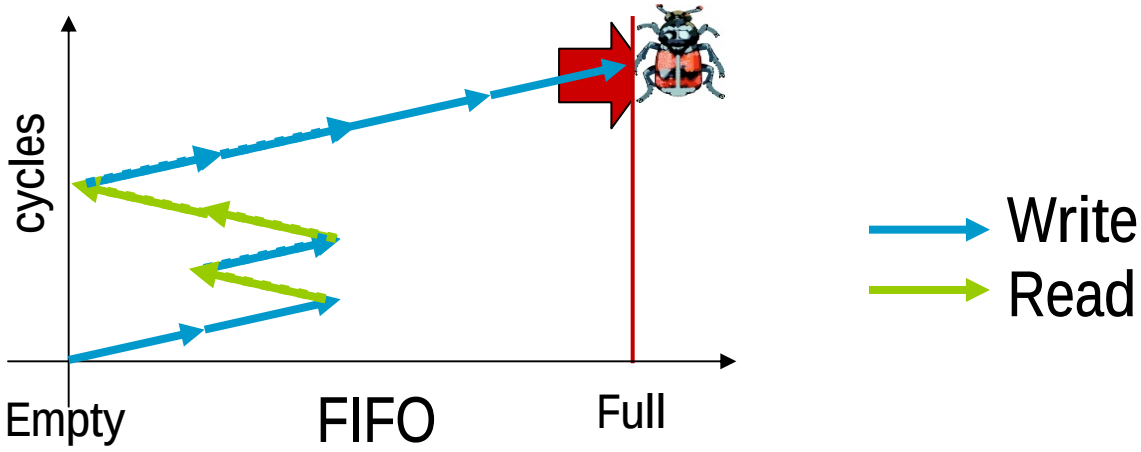
反例を短くしたい

- ランダムシミュレーションから得られる反例は、一般的に非常に長く、何故悪いのか理解できない

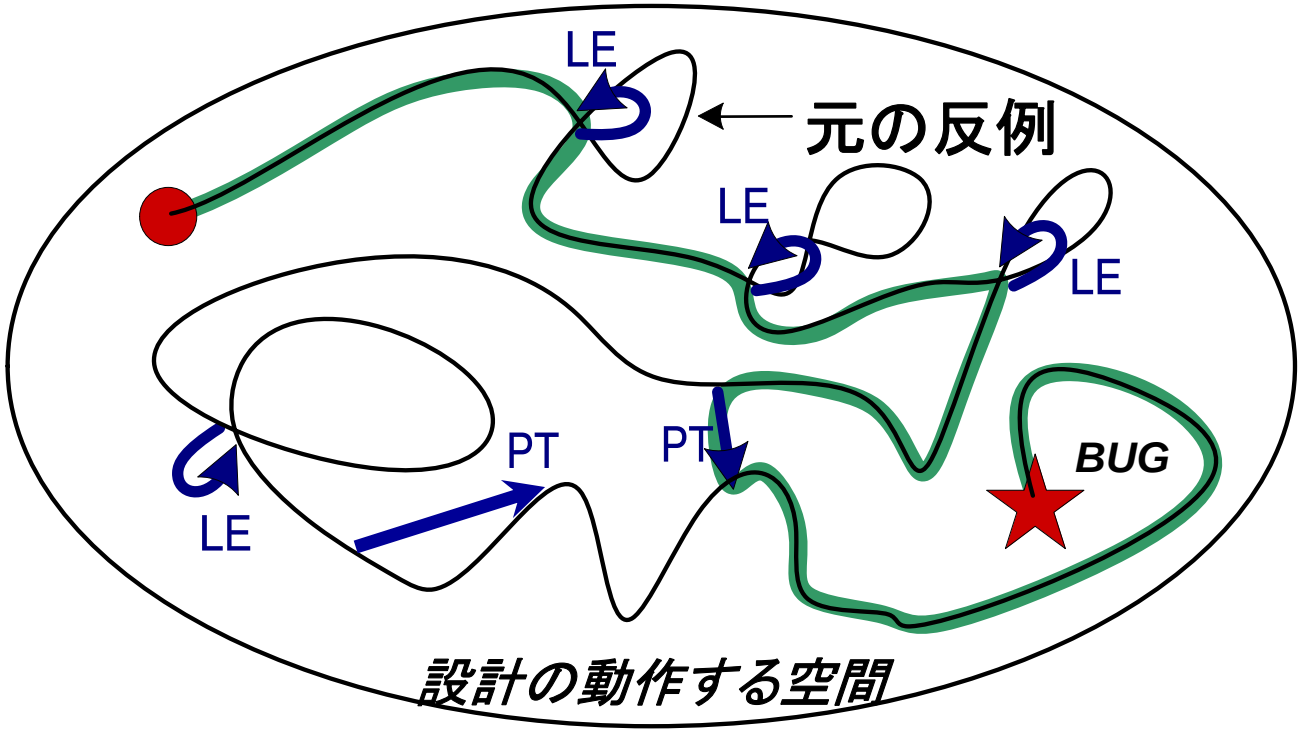




- 意味から考えて、短くなるはず
  - FIFOが一杯になった時に発生するバグ



形式的手法(記号シミュレーション)で反例を短くする



- ループを取り除く
- ショートカットを発見的に探索