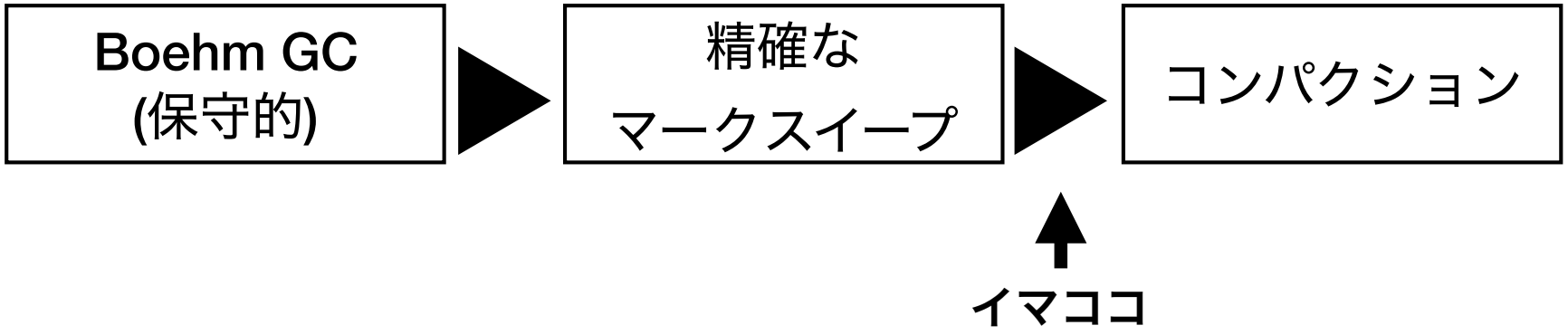
精確なガーベージコレクションのための局所変数登録誤りの発見

鵜川 始陽　　藤本 太希

高知工科大学
KOCHI UNIVERSITY OF TECHNOLOGY

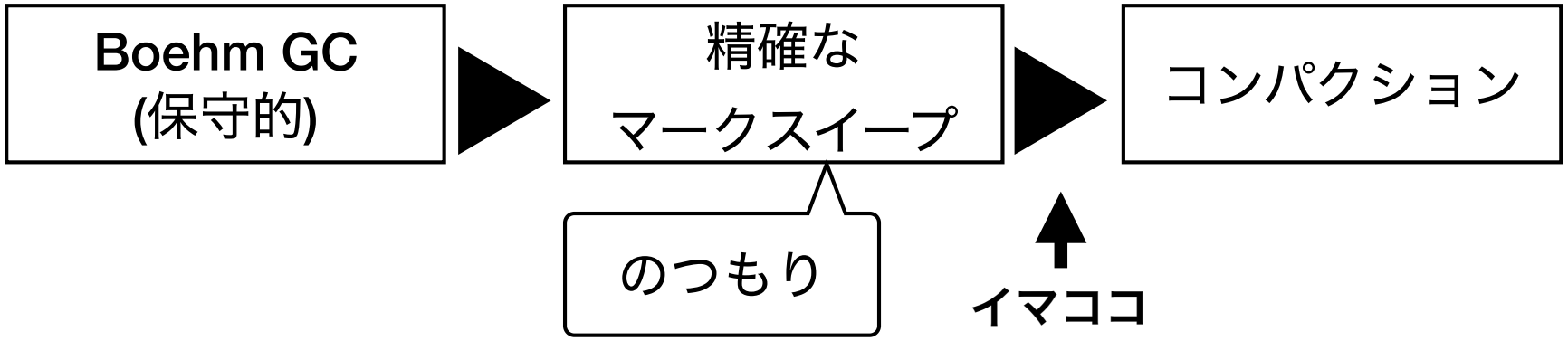
背景

- eJSVM: JavaScript仮想機械（VM）
  - C言語で記述されたVM
- ガーベージコレクション（GC）の変遷



背景

- eJSVM: JavaScript仮想機械（VM）
  - C言語で記述されたVM
- ガーベージコレクション（GC）の変遷



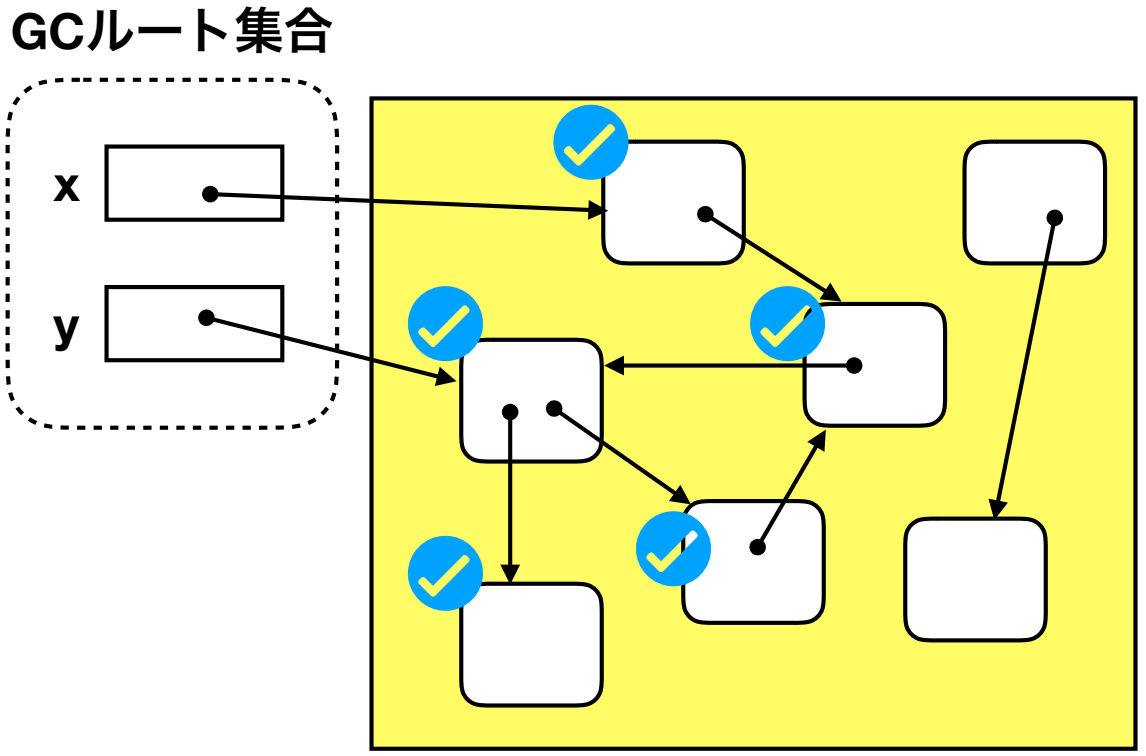
- GCのバグ発覚

この話の概要

- eJSVMのデバッグ体験
- GCのバグを探すのに Coccinelle というツールを使った
- 制御フローグラフに対するパタンマッチが有効だった

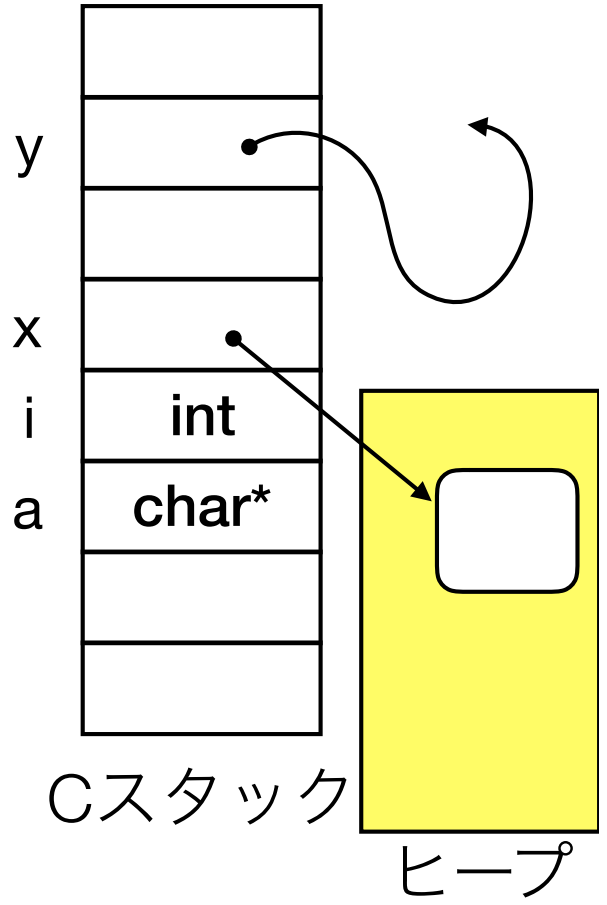
GCルート集合

- GCはポインタをたどって生きているオブジェクトを探す
- 探索の起点がGCルート
  - 大域変数
  - 局所変数
  - C言語の変数



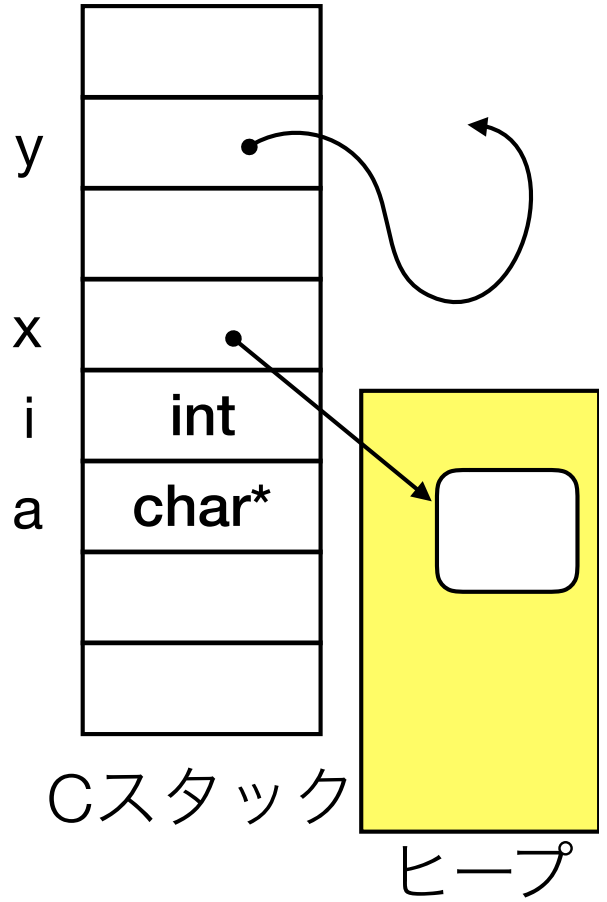
C言語の局所変数

- スタック上に存在
- GCはスタックの型が分からない
- 変数が初期化済みかどうかも分からない



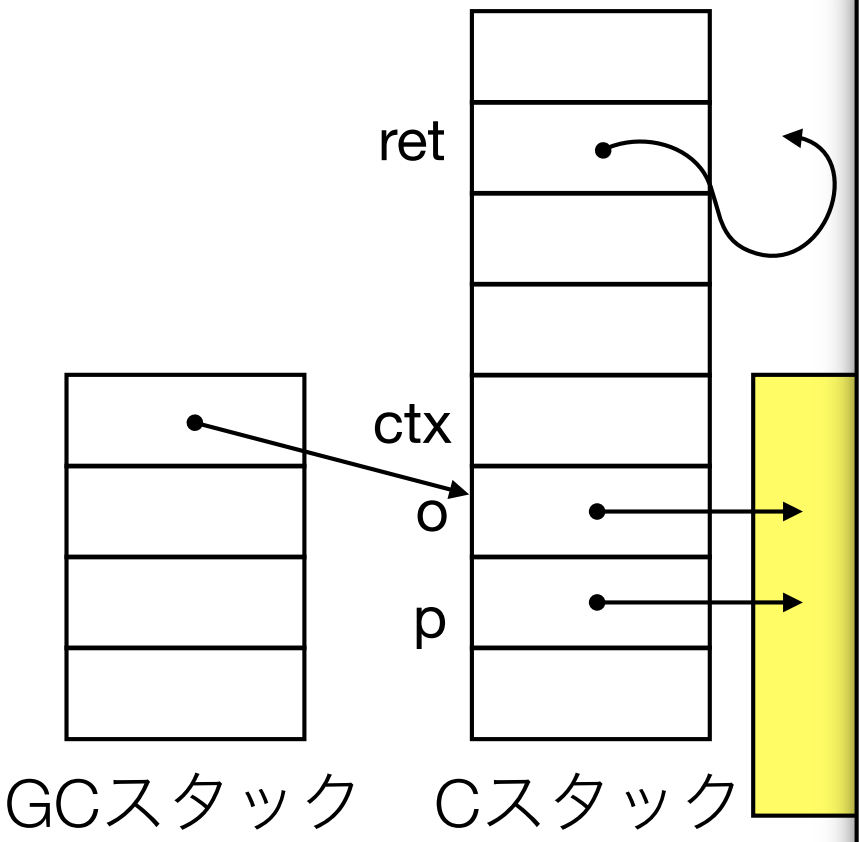
局所変数の探し方

- 保守的GC
  - オブジェクトを移動できない
- スタックマップ
  - 実装が大変
- プログラマが明示 ← eJSVMの実装
  - 最初は手軽
  - コードの品質を保つのが大変



GC_PUSHとGC_POP

- ヒープへのポインタを持つ変数の**アドレス**をGCスタックに積む




```
get_obj_prop(C *ctx, JV o, JV p){
  JV ret;
  if (!is_string(p)) {
    GC_PUSH(o); // gc_push(&o)
    p = to_string(ctx, p);
    GC_POP(o);  // gc_pop()
  }             // 引数はデバッグ用

  do {
    if (get_prop(o, p, &ret))
      return ret;
  } while (get__proto__(o, &o));
  return JS_UNDEFINED;
}
```

GC_PUSH挿入の条件

- GCが起こる可能性がある
- JV型の変数
- GCまでに初期化されている
- GC後にその値が使われる

GC前にGC_PUSH

```
get_obj_prop(C *ctx, JV o, JV p){
  JV ret;
  if (!is_string(p)) {
    GC_PUSH(o); // gc_push(&o)
    p = to_string(ctx, p);
    GC_POP(o);  // gc_pop()
  }             // 引数はデバッグ用

  do {
    if (get_prop(o, p, &ret))
      return ret;
  } while (get__proto__(o, &o));
  return JS_UNDEFINED;
}
```

GC_PUSH挿入の条件

- GCが起こる可能性がある
- JV型の変数
- GCまでに初期化されている
- GC後にその値が使われる

コンテキスト（C型）の引数をとる関数の中でしか起こらない

GC前にGC_PUSH

```
get_obj_prop(C *ctx, JV o, JV p){
  JV ret;
  if (!is_string(p)) {
    GC_PUSH(o); // gc_push(&o)
    p = to_string(ctx, p);
    GC_POP(o);  // gc_pop()
  }             // 引数はデバッグ用

  do {
    if (get_prop(o, p, &ret))
      return ret;
  } while (get__proto__(o, &o));
  return JS_UNDEFINED;
}
```

堅実な挿入方法

- 全てのJV型変数は宣言と同時に初期化
- 全てのJV型変数をスコープに入ると同時にGC_PUSH
- パフォーマンス悪い?
- この方法でもミスする

```
get_obj_prop(C *ctx, JV o, JV p){
  JV ret = NULL;
  GC_PUSH3(o, p, ret);
  if (!is_string(p))
    p = to_string(ctx, p);
  do {
    if (get_prop(o, p, &ret))
      return ret;
  } while (get__proto__(o, &o));
  GC_POP3(o, p, ret);
  return JS_UNDEFINED;
}
```

解きたい問題

- GC_PUSHの挿入漏れを探す
  - 手動でGC_PUSHを挿入してあるプログラムが対象

- GCが起こる可能性がある
- JV型の変数
- GCまでに初期化されている
- GC後にその値が使われる

GC前にGC_PUSH

方針

- テスト　→ 不完全，バグの特定が難しい
- 静的解析
  - **実行経路（パス）に関するパタンマッチ**
    - **Coccinelle**
  - 構文的なパタンマッチ　→ 向かない
    - ASTgrep, ASTMatcher, PMD, …
  - 抽象実行，型，難しい解析，… → 使うコストが大きい？

なぜパスに関するパタン?

- GCが起こる
- JV型の変数
- **GCまで**に初期化されている
- **GC後**にその値が使われる

GC前にGC_PUSH

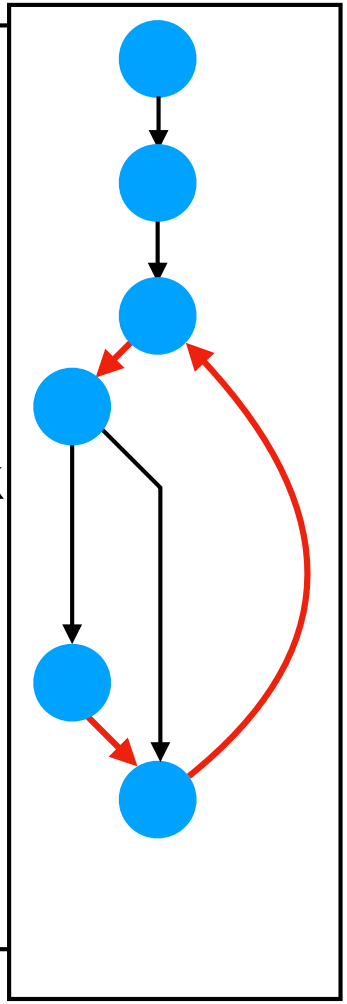
```
JV a;
…
a = args[i];
…
while(k < len){
  if(has_array_elem(a,k)){
    kv = cint_to_fixnum(k);
    nv = cint_to_fixnum(n);
    e = get_array_prop(a,kv);
    set_array_prop(ctx,b,nv,e);
  }
  n++; k++;
}
```

なぜパスに関するパタン?

- GCが起こる
- JV型の変数
- **GCまで**に初期化されている
- **GC後**にその値が使われる

GC前にGC_PUSH

```
JV a;
…
a = args[i];
…
while(k < len){
  if(has_array_elem(a,k))
    kv = cint_to_fixnum(k
    nv = cint_to_fixnum(n
    e = get_array_prop(a,
    set_array_prop(ctx,b,
  }
  n++; k++;
}
```

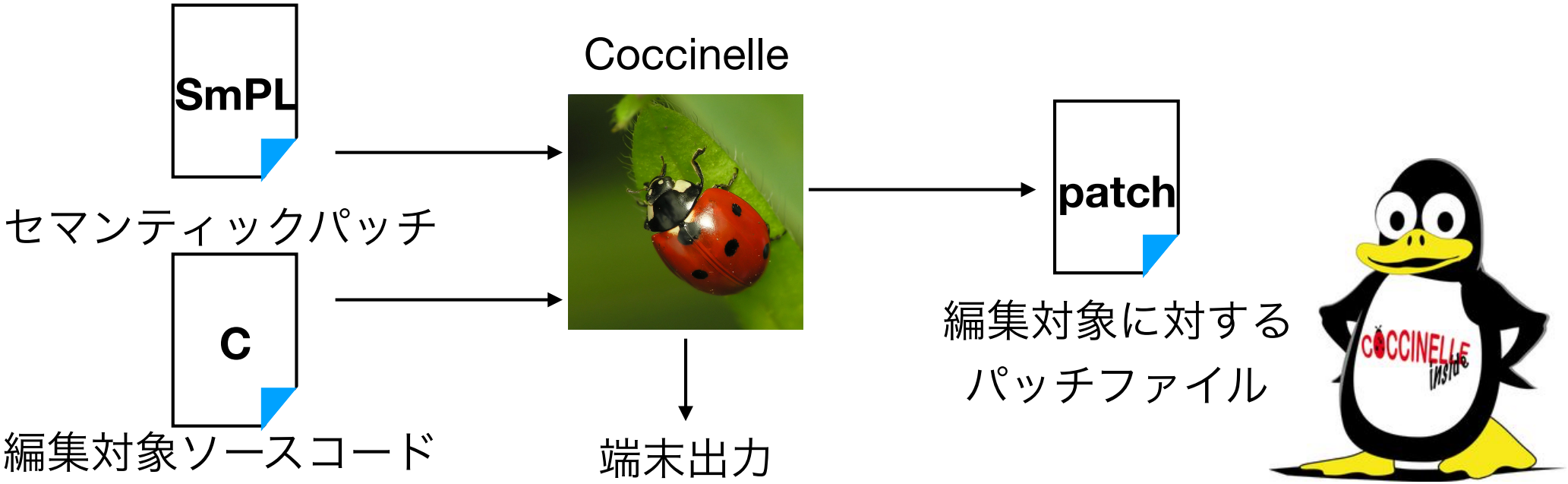


目次

- 背景と目的
- Coccinelle
- GC_PUSH漏れ発見のパタン作成
- 評価とケーススタディ
- 議論

Coccinelle [Muller, Lawall, et al. 2009]

- プログラムのパタンマッチと編集を行うソフトウェア
- Linuxカーネルのデバッグやソフトウェア進化に利用
  - バグの発見，ライブラリAPIの変更に伴うコードの修正
- 現在でも開発が続いている [Kang et al. (ECOOP 2019)]



Coccinelleの利用例

- Linux[Lawall et al. 2009, Palix et al. 2014]
- OpenSSL [Lawall et al. 2010]
- GCのバグ発見に利用した例はない

セマンティックパッチ

- patchコマンドに入力するdiffに似ている（開発者の見解）
- 意味的なパタンマッチをする
  - 空白や括弧の有無などは無視
  - 構文要素や出現位置をメタ変数で抽象化
  - 「...」で途中を省略
    （省略部分の条件を指定可）
  - 同じ意味の構文を同一視
    例) x == NULL と NULL == x

```
@@
expression E,E1;
@@

if (
-    E == NULL
+    IS_ERR(E)
    ) {
    <+... when != E = E1
       PTR_ERR(E)
       ...+> }
```

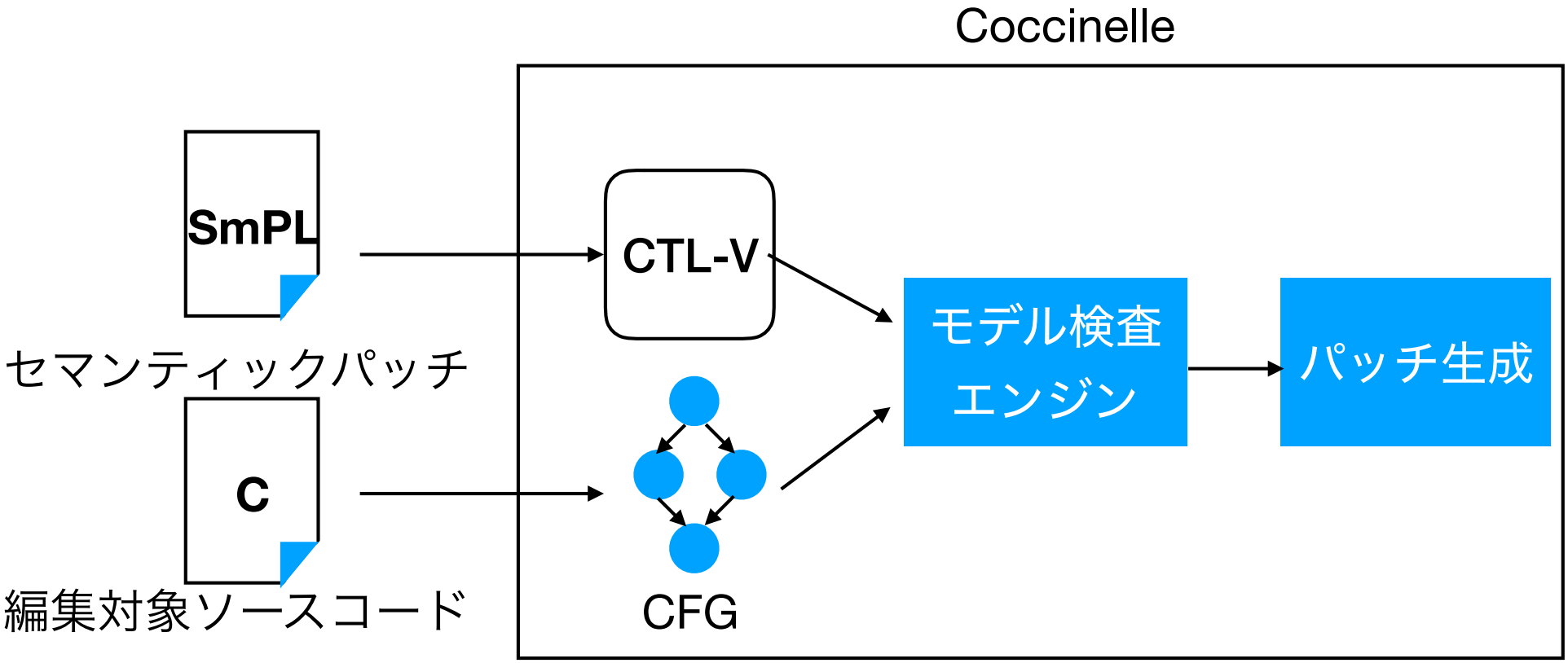
A semantic patch correcting a case where a value is compared to NULL and subsequently passed to PTR_ERR

http://coccinelle.lip6.fr/rules/ より

Coccinelleの内部

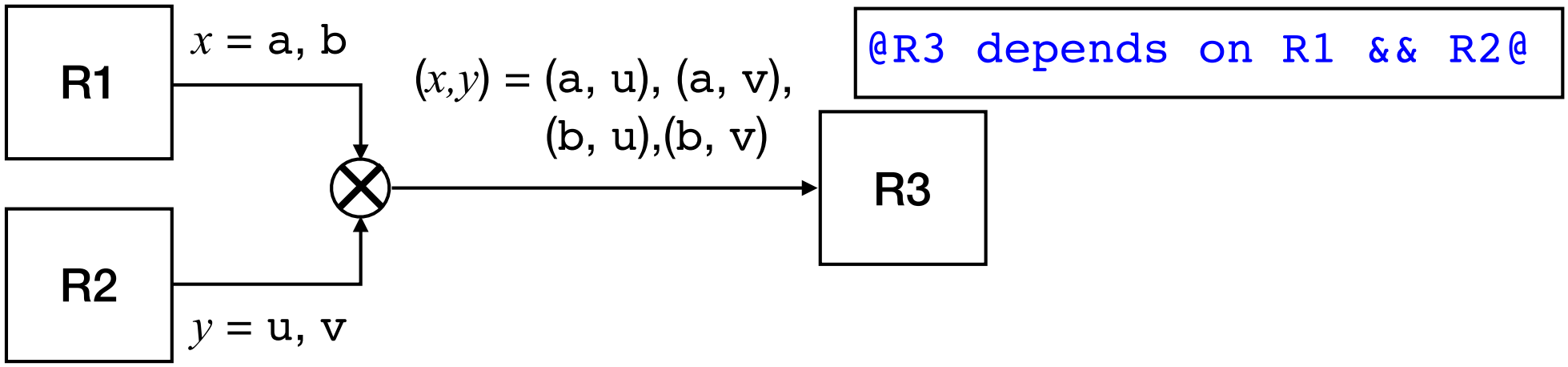
制御フローグラフ（CFG）に対するパタンマッチ

- セマンティックパッチをメタ変数付き計算木論理（CTL-V）式に変換してモデル検査 [Brunel et al. 2009]



ルールの組み合わせ

- 複数のルールでパッチを構成できる
  - 後続のルールが先行ルールに依存する
  - マッチしたメタ変数を後続のルールで利用できる
- 複数のルールに依存するルールも作れる
- 先行ルールに「マッチしない」という条件も指定できる



目次

- 背景と目的
- Coccinelle
- GC_PUSH漏れ発見のパタン作成
- 評価とケーススタディ
- 議論

GC_PUSH漏れバグ

GC_PUSH挿入の条件

- GCが置こる
- JV型の変数
- GCまでに初期化されている
- GC後にその値が使われる

GC前にGC_PUSH

バグの条件

1.GCが起こる

2.JV型の変数

3.GCまでに初期化されている

4.GC後にその値が使われる

5.GC前にGC_PUSH**がない**

「GCまでに初期化されている」は条件に含めない

- GC後に初期化して使う場合
  - GCの時の値は使われない

    ⇒ 条件4にマッチしない

    ⇒ 条件3は不要
- GC後にも初期化せず使われる場合
  - GCに関係ないバグ

    ⇒ gccが検出

    ⇒ 条件3は不要

1.GCが起こる

2.JV型の変数

3.GCまでに初期化されている

4.GC後にその値が使われる

5.GC前にGC_PUSHがない

条件の詳細な記述

1.GCが起こる

2.JV型の変数

3.GCまでに初期化されている

4.GC後に**その**値が使われる

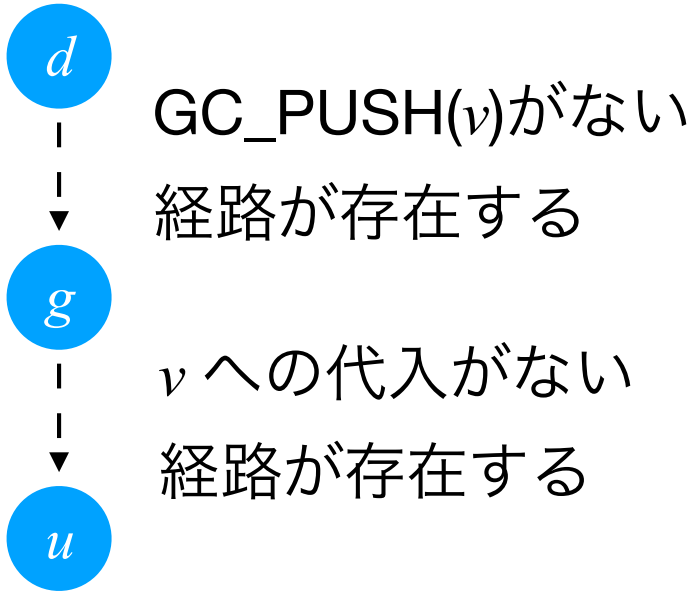
5.GC前にGC_PUSHがない

forall GCが起こる位置 g

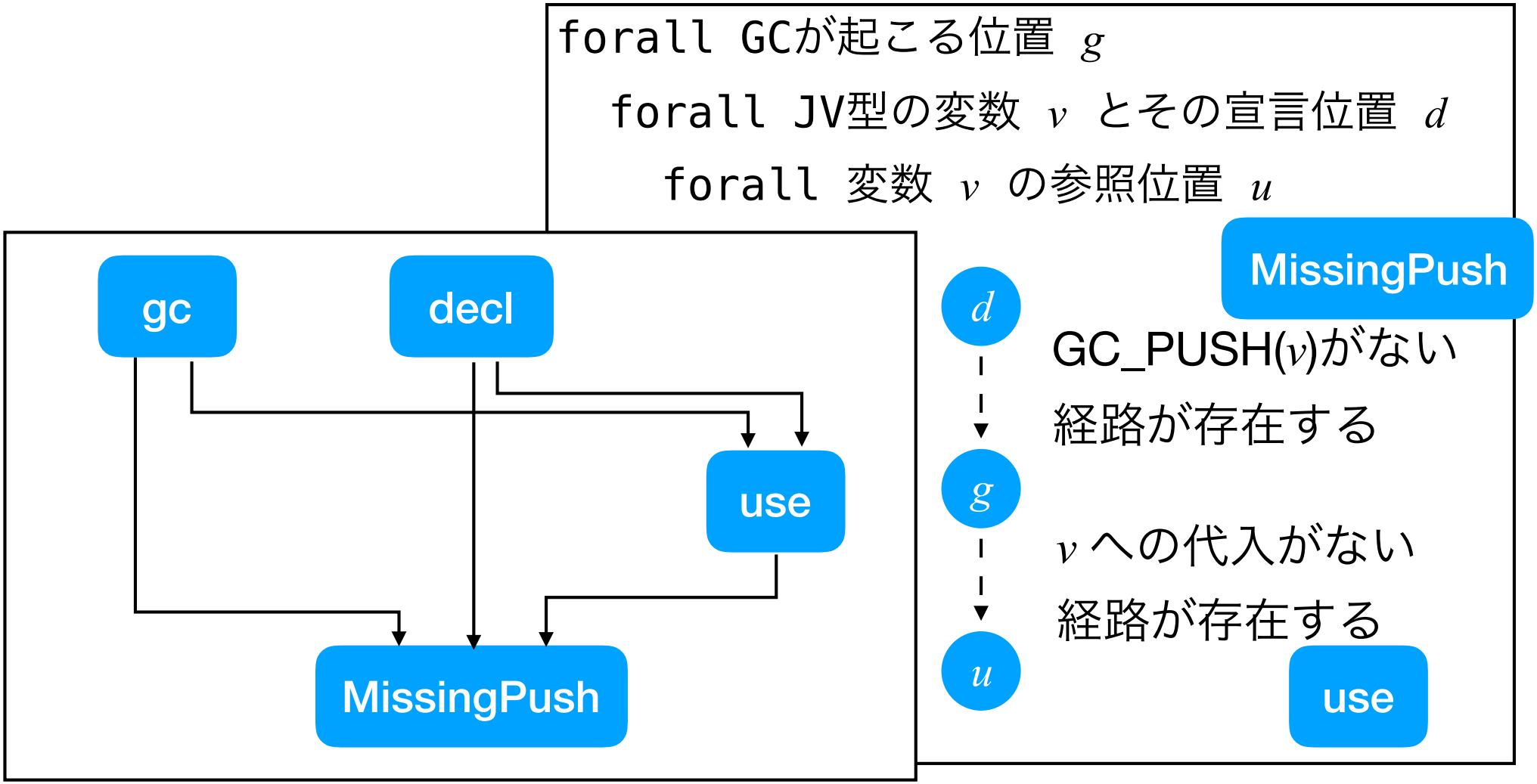
  forall JV型の変数 v とその宣言位置 d

    forall 変数 v の参照位置 u

      次を探す:



セマンテックパッチの構成



use と MissingPush

```
@use depends on gc&&decl@
identifier decl.v;
expression e, gc.gc_fun;
position gc.gc_p, use_p;
@@
gc_fun@gc_p
... when != v = e
    when exists
v@use_p
```

vへの代入がない
経路が存在する

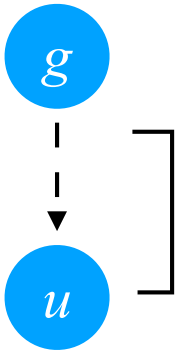
```
@MissingPush depends on use@
identifier pre.push, decl.v;
expression e, gc.gc_fun;
position gc.gc_p,decl.decl_p;
type decl.T;
@@
(
T v@decl_p;
|
T v@decl_p = e;
)
... when != GC_PUSH(v)
    when exists
gc_fun@gc_p
```

GC_PUSH(v)がない
経路が存在する

偽陽性の除去

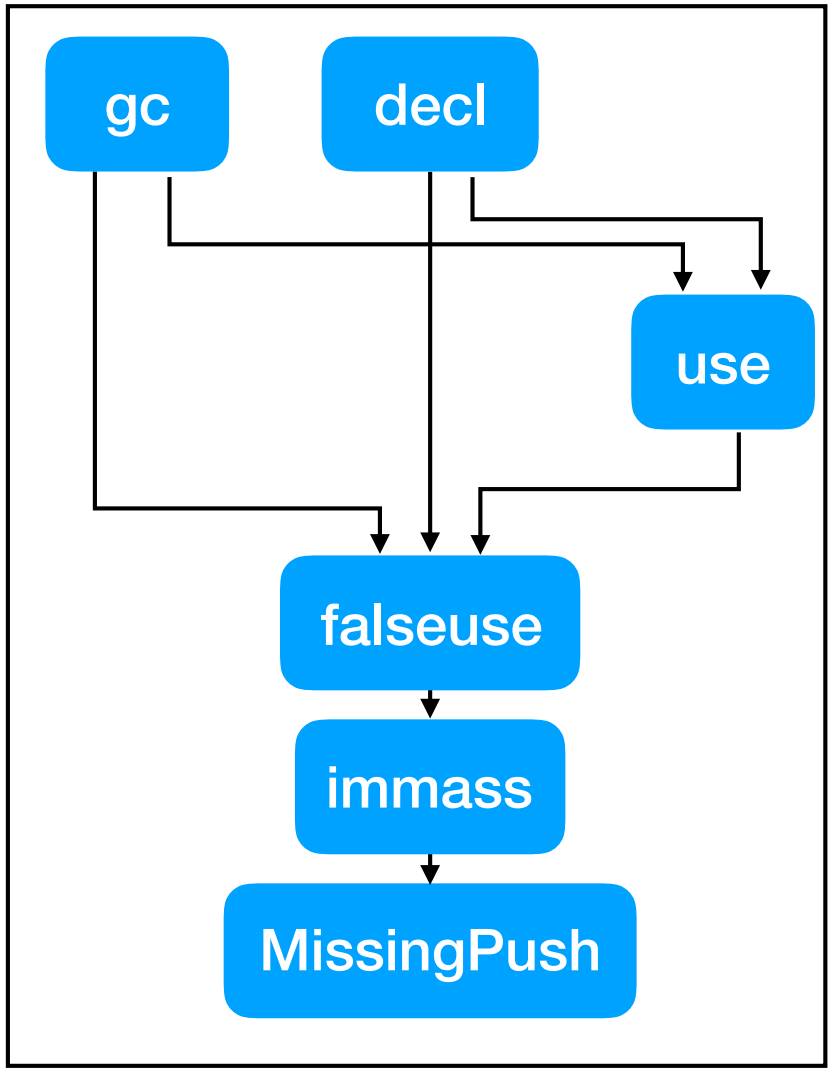
- 最初のパッチでは偽陽性が多い
  - vへの代入がvの参照にマッチする
  - GC_PUSH(v)がvの参照にマッチする
  - GCを起こす関数の呼出しがある文の中でのvへの代入が考慮されていない
- ルールを追加してフィルタする

```
g(ctx); // GC
v = …
if (…) {
    GC_PUSH(v);
    h(ctx); // GC
    GC_POP(v);
}
```



v への代入がない経路が存在する

フィルタ




```
@falseuse …@
(
v@use_p = e
|
GC_PUSH(v@use_p)
|
GC_POP(v@use_p)
)

@immass …@
v = <+... gc_fun@gc_p …+>

@MissingPush depends on
 use && !falseuse && !immass@
```

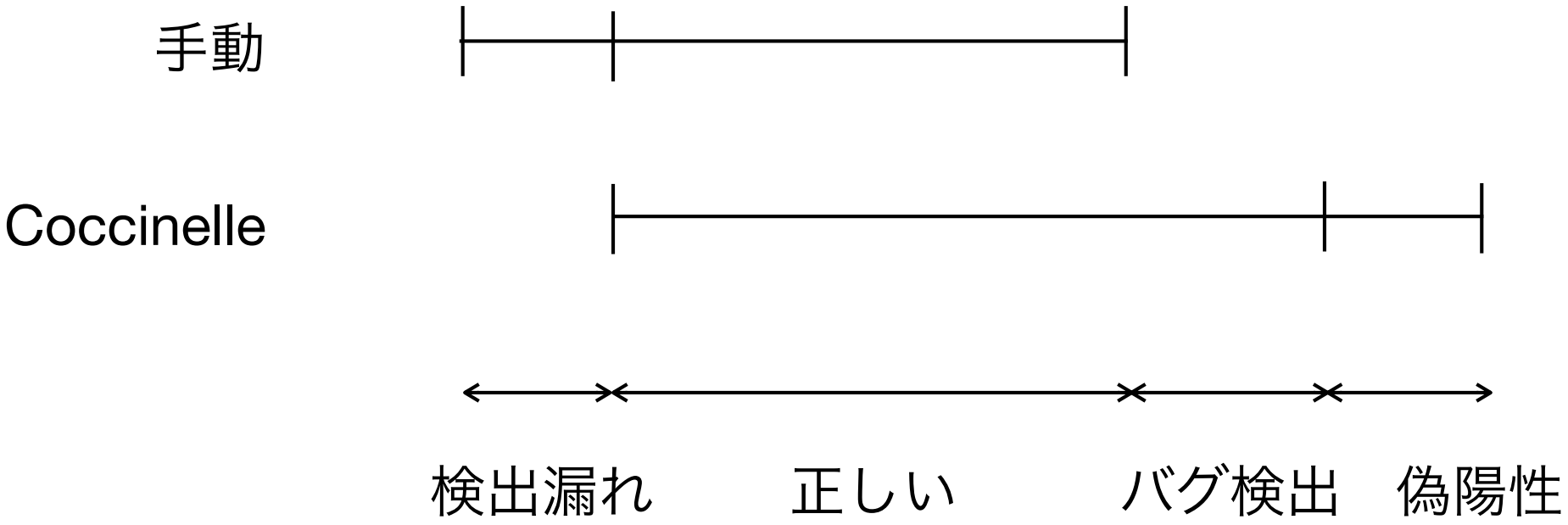
目次

- 背景と目的
- Coccinelle
- GC_PUSH漏れ発見のパタン作成
- 評価とケーススタディ
- 議論

評価

- eJSVMのソースコードを利用
  - ファイル数20　（発表資料にある unix.c は間違い）
  - 行数 9459 行（コメント，空白含む）
- 手動で挿入していたGC_PUSHを取り除いてGC_PUSH漏れとして検出できるか調査

結果の分類



※ 真の正解は誰にも分からない

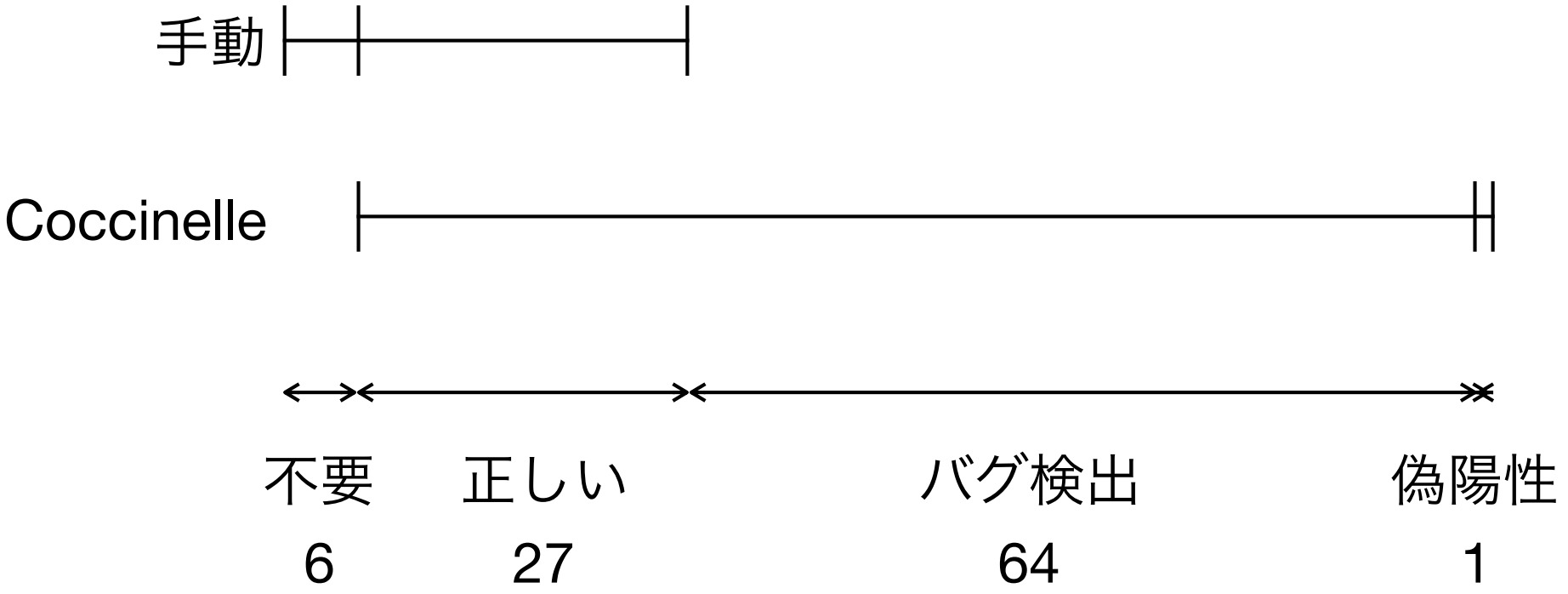
結果

- 検出漏れはなかった

結果

- 検出漏れはなかった
- 不要なGC_PUSHがあった

結果



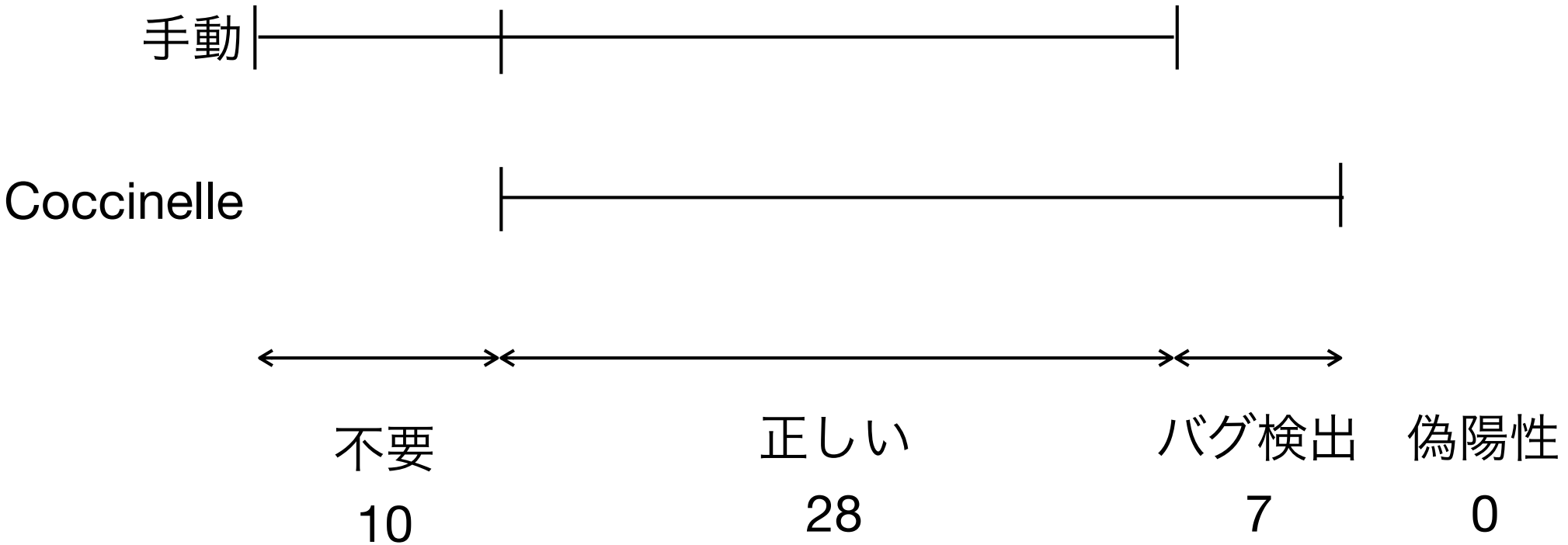
- 検出漏れはなかった
- 不要なGC_PUSHがあった

- 偽陽性は1つだけ
- eJSVMのコードの品質が悪い?

評価（2）

- 他人のソースコードを利用
  - C言語によるAVL木の実装
  - 新しいGCの実験用コード
  - ファイル1個，368行
- 全ての関数がGCを起こす可能性がある
  - コンテキストは大域変数

結果



- 不要なGC_PUSHは存在した
- バグも検出された

目次

- 背景と目的
- Coccinelle
- GC_PUSH漏れ発見のパタン作成
- 評価とケーススタディ
- 議論

有効性

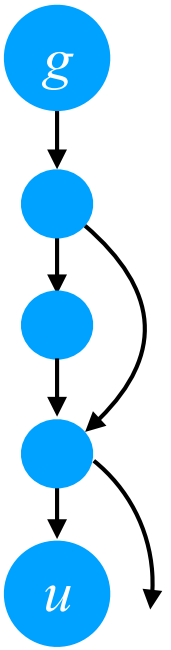
- CFG上のパタンマッチが有効だった
  - GC_PUSHを追加する条件の特徴
    - 関数内で閉じている
      （通常のリソース管理は閉じていない）
    - パスに関する条件
  - CFGを作るライブラリを使って専用ツール化の検討
    - Coccinelleを使うのに比べ改善される点が少ない
    - 問題ごとに設定が異なるので割に合わなさそう

限界: データに依存する場合

- データに依存して分岐が制限される場合は偽陽性になる

```
if (low_exists)
  low_val = get(ctx, low_index);
if (high_exists)
  high_val = get(ctx, high_index);


if (low_exists && high_exists) {
  set(ctx, low_index, high_val);
  …
```



限界: PUSHとPOPの対応

- GC_PUSHとGC_POPが精確に対応しているかは検査できなかった
- eJSVMではGC_POPの引数はデバッグ用なので間違っていても動く

```
GC_PUSH(a);
GC_PUSH(b);
f(ctx);
GC_POP(a);
GC_POP(b);
```

GC_PUSH自動挿入

- 現在は自動化していない
  - どこにGC_PUSHするかは人間が判断
  - それなりの個所に挿入することはできるがそれが正しいかは確認が必要
- 次の情報を提示することで十分実用的
  - 変数の宣言位置
  - GCが起こる可能性がある位置
  - 変数が参照される位置

まとめ

- GCのためのソースコードの修正漏れを静的検査により探した
- 検査にはCoccinelleを使った
- 制御フローグラフに対するパタンマッチが有効だった
- eJSプロジェクトで現在も使っている