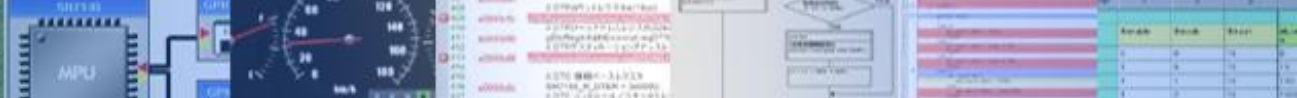




# MC/DCカバレッジ測定機能 (オプション機能)

※ CasePlayer2との連携機能



# MC/DC計測のためにはコード挿入が必須

- 実コードそのままの状態では複合条件式の評価は不可能
  - MC/DCが評価対象とするのは複合条件式内の各条件式の実行論理
  - 原理的に、どのようなツールでも、各条件式を個別に評価することはできない
- MC/DC計測では複合条件式の構造を変更する必要がある
  - 各条件式を独立して評価できるようにコードの構造を変更する
- MC/DC計測用のフックコードを挿入した「埋め込みコード」で計測
  - CasePlayer2により、評価対象の関数から自動生成される

**if( ( x>10 && y>20 ) || z>30 )** ←元の複合条件式



フックコードを挿入

**if(Hook(Hook(x>10) && Hook(y>20) ) || Hook(z>30) )**

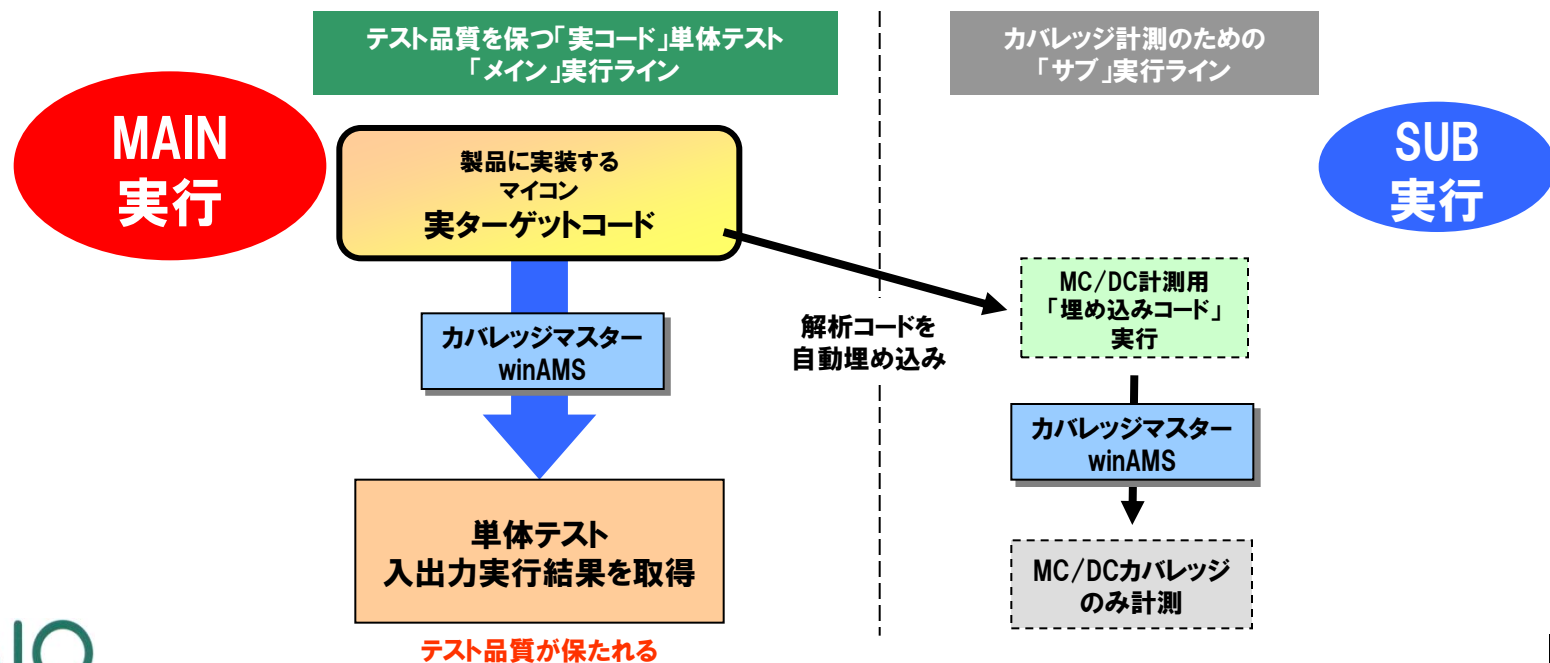
↑ 各条件式の論理をHook()関数に送り、  
Hook()の中で実行状態を記録する構造に変更する



# 実コード計測と埋め込みコード計測を併用

- 複合条件計測のために フックコードを埋め込み MC/DC計測実行
- MC/DCカバレッジのみ「埋め込みコード」から計測  
 実行結果(期待値比較)は「実コード」から取得し評価
  - 実行結果は実コードから取得することで テスト品質を保つ

←重要!





# MC/DCカバレッジ測定方法のポイント

## ■ 実コードによるテスト品質を維持したまま MC/DCを計測



## ■ テスト実行結果(出力)は「実コード」実行結果から取得

- 出力結果は ソースに手を加えない「実コード」から取得(従来の標準機能と同じ)

## ■ MC/DCカバレッジは解析コードを挿入した「埋め込みコード」から取得

- MC/DCカバレッジは、複合条件に分離するために、解析コードを挿入して実行
- MC/DC計測用「埋め込みコード」の実行のために ビルドプロジェクトを自動複製
- ※C0、C1カバレッジも、埋め込みコードから取得可能
  - 最適化によるコード構造の変化により、実コードではカバレッジ計測ができない場合に適用

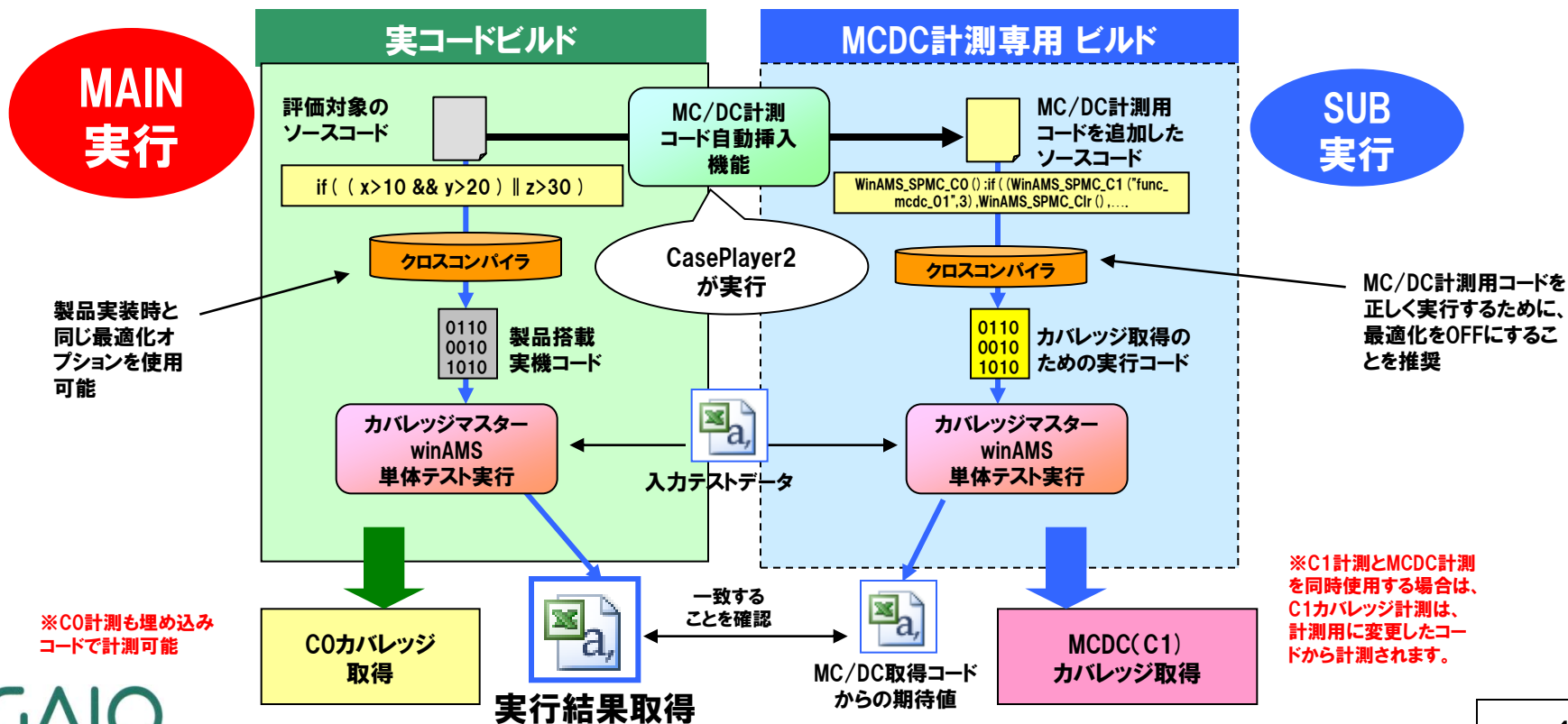
## ■ 「実コード」実行結果と「埋め込みコード」実行結果を比較

- MC/DC条件解析コードを追加しても、関数出力値に相違が無いことを確認
- 相違が無いことを持って MC/DCカバレッジ結果の確からしさを検証

# MC/DCカバレッジ測定機能：詳細処理フロー

## ■ 「実コード」と「埋め込みコード」を並列実行

- MDCDC計測には、複合条件式を分解する解析コードを自動挿入し実行
- 入出力テスト、期待値評価には、「実コード」から取得



# MC/DCカバレッジ測定機能：解析コード挿入

## ■ 各条件式を「フック関数」に送るコードに変更（コード挿入イメージ）

### MC/DCカバレッジ 評価を行う 対象関数

```
//
int func_01( int x, int y, int z )
{
    /* 複合条件設定 */
    if( ( x>10 && y>20 ) || z>30 )
    {
        gb_result.data = x+y-z;
        gb_result.ret_code = TRUE;
        return TRUE; // 条件TRUEの時、TRUEを返す
    }
    else
    {
        gb_result.data = z+y-x;
        gb_result.ret_code = FALSE;
        return FALSE; // 条件FALSEの時、FALSEを返す
    }
}
```

コード挿入

### カバレッジ計測のために自動生成されたフック関数挿入後の関数

```
//
int HOOKED_func_01( int x, int y, int z )
{
    /* カバレッジ条件式 Hook Routine 挿入 */
    if( Hk_Cr(), Hk_R(Hk_E(x>10) && Hk_E(y>20) ) || Hk_E(z>30) )
    {
        Hk_C1cvr(1); // 実行マーキングフックコード
        gb_result.data = x+y-z;
        gb_result.ret_code = TRUE;
        return TRUE; // 条件TRUEの時、TRUEを返す
    }
    else
    {
        Hk_C1cvr(0); // 実行マーキングフックコード
        gb_result.data = z+y-x;
        gb_result.ret_code = FALSE;
        return FALSE; // 条件FALSEの時、FALSEを返す
    }
}
```

各条件式を  
別々に実行

### ツールのカバレッジ計測のためのフック関数

```
int Hk_Cr( void )
{
    // フックの情報クリア
}

int Hk_C1cvr( int path_no )
{
    // 実行マーキング
}

int Hk_E( int condition )
{
    // MCDC用論理式の評価
    return condition;
}

int Hk_R( int condition )
{
    // 論理式の判定
    return condition;
}
```

# C0、C1、MC/DCカバレッジ表示

**C0、C1カバレッジ (実機コードから取得)**

196		5	int func4( int code )
197		5	{
198		5	int return_value=FALSE;
199		5	int i;
200		5	
201		5	if( gb_a > 10 )
202	T/F	5	{
203		2	if( gb_b > 20 && gb_c > 30 )
204	T/F	2	{
205		1	gb_out = 0;
206		1	}
207		1	else
208		1	{
209		1	gb_out = -1;
210		1	}
211		2	return_value = FALSE;
212		2	
213		2	else
214		8	{
215		8	switch( code )
216	3/4	8	{
217		1	case 1:
218		1	gb_out = 1;
219		1	break;
220		1	case 2:
221		1	gb_out = 2;
222		1	break;
223		1	case 3:
224		0	gb_out = 3;
225		0	break;
226		0	default:
227		1	gb_out = -1;
228		1	break;
229		1	}
230		3	return_value = FALSE;
231		3	
232		3	}
233		3	

C1カバレッジ  
(実行分岐数)

switch文のcase3が  
実行されていない  
→C1カバレッジNG

C0カバレッジ  
(各行の実行数)

表示  
切替

**MCDCカバレッジ結果 (フックコード付きから取得)**

201		5	if( gb_a > 10 )
202	T/F	5	[MC/DC t/f] gb_a>10
203		2	{
204	T/F	2	if( gb_b > 20 && gb_c > 30 )
		2	[MC/DC t/f] gb_b>20
		2	[MC/DC t/] gb_c>30
205		1	gb_out = 0;
206		1	}
207		1	else
208		1	{
209		1	gb_out = -1;
210		1	}
211		2	return_value = FALSE;
212		2	
213		2	}

MCDCカバレッジ  
(t=TRUE f=FALSE)

※最後の'gb\_c>30'の条件文のFALSE  
ケースが実行されていないことを示す

実行回数が0  
→C0カバレッジがNG