

C/C++エンジニア必見！コード品質向上 & 効率的ランタイムエラー根絶テクニック

MathWorks Japan

アプリケーションエンジニアリング部

田中 康博

ランタイムエラーのもたらす危険性

Ariane 5

GNC システムの誤動作

\$500M 積荷

+ \$7B 開発費

\$7.5B 損失

オーバーフロー

USS Yorktown

推進システムが繰り返しシャット
ダウンして制御不能になる

ゼロ除算

Therac 25

放射線の過剰摂取

データ競合

オーバーフロー

いつ出荷するのが安全ですか？

全てのバグの **40%**

ランタイムエラー

- IBMによる調査

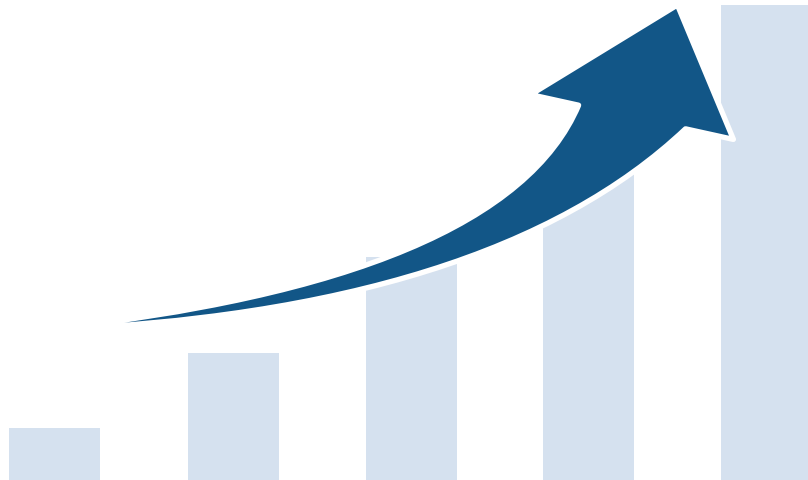
全ての医療機器の **33%**

ソフトウェア不具合により
リコール

(1999年から2005年までに米国で販売)
- U. of Patras (Greece) による調査

ソフトウェア不具合の早期発見で開発コスト低減とリスク削減

ソフト複雑度 & 開発負荷の増大



開発の流れ

従来のソフトウェア開発の特徴

- 安全性の認証とテストはソフトウェア開発コストの大部分を占める
- 不具合修正コストが高い後工程にテストが集中



静的コード解析はシフトレフトするための最も簡単で最善の方法

Polyspace による静的解析とテスト

- ✓ C
- ✓ C++
- ✓ Ada

Method:

形式手法に基づくセマンティック解析 (抽象解釈)

Tool Qualification & Certification:

- ✓ DO-178C/DO-331
- ✓ ISO 26262, IEC 61508, IEC 62304, ISO 25119, EN 50128, EN 50657

C/C++



C/C++

手書きコード

自動生成コード
(Simulink連携)

Goal:

- ✓ ランタイムエラーを検出する
(ゼロ除算、オーバーフローなど)
- ✓ コーディング規則の準拠違反を検出する
(MISRA C, CERT Cなど)
- ✓ コードメトリクスを測定する
- ✓ ランタイムエラーの有無を証明する
- ✓ 単体テストを行う **R2023b**

Polyspace ユーザー事例

日産自動車、ソフトウェアの信頼性を向上

「Polyspace 製品によって、高レベルなソフトウェアの信頼性を確保することができます。これは、業界内の他のツールにはできないことです。」

— 菊池光彦氏, 日産自動車



日産 フェアレディZ

日産自動車の最優先事項は品質です。日産自動車株式会社 ソフトウェア品質グループのリーダー、菊池光彦氏は次のように説明します。「日産は顧客に対して重大な責任を担っていますので、当社の車両には厳格な品質基準が設けられています。これらの基準は、車両搭載の組み込みソフトウェアにも適用されます。品質評価プロセスが全ての電子制御ユニット (ECU) 上のソフトウェアに効率的に適用されるよう、ソフトウェア品質グループは全てのサプライヤーと協働して、プロジェクトの開始から節目ごとにソフトウェアのレビューを行っています。またソフトウェア品質グループは、ソフトウェア品質について日産の役員に直接報告しています。」

課題

ソフトウェア品質を向上させるために、発見が困難なランタイムエラーを特定する

ソリューション

MathWorks のPolyspace製品を使用して、日産とサプライヤーのコードを包括的に解析する

結果

- サプライヤーのバグを検出して評価
- ソフトウェアの信頼性が向上
- 日産のサプライヤーが Polyspace 製品を採用

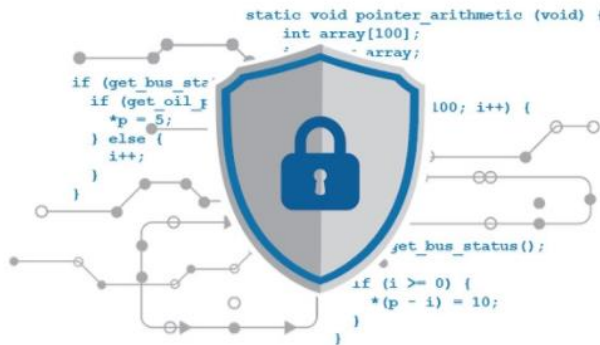
使用製品

- Polyspace Bug Finder
- Polyspace Code Prover

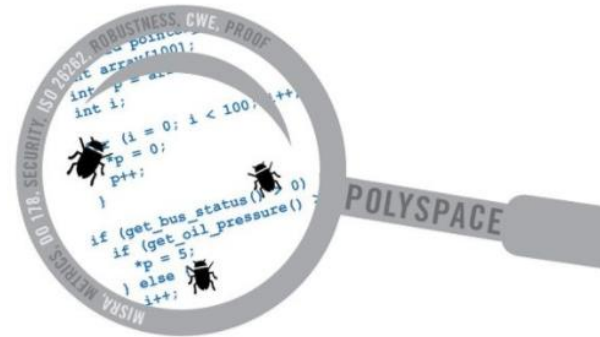
» 日産自動車に関する詳細はこちら

Polyspace Bug Finder でバグの作りこみ抑制と早期発見する

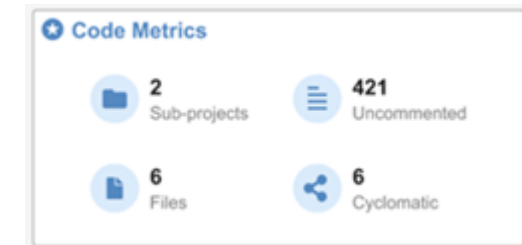
コーディングルール
の準拠性を確認する



バグとセキュリティ
脆弱性を検出する



コードメトリクス
を測定する



コーディングルール

- MISRA C:2004
- MISRA C:2012
- MISRA C++:2008
- JSF C++
- AUTOSAR C++14

セキュアコーディングルール

- CERT C/C++
- ISO 17961
- CWE
- MISRA C:2012 Amendment 1

暗号化

- 脆弱な暗号アルゴリズム

データフロー

- 読取りのない書込み

プログラミング

- 宣言の不一致

静的メモリ

- NULLポインター

動的メモリ

- メモリリーク

同時実行

- デッドロック

プロジェクトメトリクス

- 再帰の数

ファイルメトリクス

- コメントなし行の数

関数メトリクス

- 循環的複雑度

など、350以上検出可能

など、30以上測定可能



独自のコーディングガイドラインをチェックする

命名規則の違反箇所を特定して可読性を向上する機会を与える

ステータス	名前	規則	パターン
1 Files			
2 Preprocessing			
3 Typedefs			
3.1	All integer types must follow the specified pattern.	Typedefs should finish by "_t"	[A-Za-z0-9]*_t
3.2	All float types must follow the specified pattern.	Typedefs should finish by "_t"	[A-Za-z0-9]*_t
3.3	All pointer types must follow the specified pattern.	Typedefs should finish by "_t"	[A-Za-z0-9]*_t

正規表現で準拠パターンを設定する

Custom 3.1 ?
All integer types must follow the specified pattern.
Typedefs should finish by "_t"
The integer type '__int8' does not match the specified pattern.

ユーザー コメント:
typedef ステートメントが「_t」で終わっていない場合の違反

コンフィギュレーション 結果の詳細

ソース

```
types.h
29 typedef unsigned int __u_int;
30 typedef unsigned long int __u_long;
31
32 /* Fixed-size types, underlying types depend
33 typedef signed char int8; 違反
34 typedef unsigned char __uint8_t;
35 typedef signed short int __int16_t;
36 typedef unsigned short int __uint16_t;
37 #ifdef __PST_16BIT_TARGET__
```

例) typedef カスタムコーディングルール

- カスタムコーディングルール：ステートメントが「_t」で終わっていること
- 準拠パターン：[A-Za-z0-9]*_t
- 違反例：__int8

コードクローンを検出する

より保守しやすいモジュールにリファクタリングする機会を与える

重複しているコード

- 全く同じコードセクションは、不必要な追加のメンテナンスが必要になる
- 他の場所を更新し忘れる可能性が高くなる

部分的に重複しているコード

- ほぼ同じコードセクションは、不必要な追加のメンテナンスが必要になる
- 他の場所を更新し忘れる可能性が高くなる

コピーして貼り付けのエラーの可能性あり

- プログラミングエラーを示している可能性がある
- コードセクションをコピーした後で1か所を更新したが、もう1か所を更新し忘れた可能性がある

```
void init(int val1, int val2, int val3, int val4,
          int val5, int val6, int val7, int val8) {
    Aboard.pin1 = val1;
    Aboard.pin2 = val2;
    Aboard.pin3 = val3;
    Aboard.pin4 = val4;
    Aboard.pin5 = val5;
    Aboard.pin6 = val6;
    Aboard.pin7 = val7;
    Aboard.pin8 = val8;
}
```

共通関数にリファクタリングして、この関数の呼び出しに置き換えることができる



○ 重複しているコード (影響: Low) ? ⓘ

This section of code seems to be duplicated in other places.

Additional Info:

Risk: Sections of code that do the same operations require unnecessary additional maintenance.

Fix: Refactor the sections of code into a dedicated function.

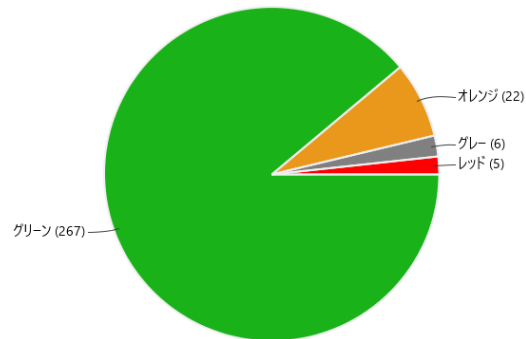
```
15 void setAllPins() {
16     int val1 = getPinVal();
17     int val2 = getPinVal();
18     int val3 = getPinVal();
19     int val4 = getPinVal();
20     int val5 = getPinVal();
21     int val6 = getPinVal();
22     int val7 = getPinVal();
23     int val8 = getPinVal();
24     Aboard.pin1 = val1; //Beginning of duplicate section #1
25     Aboard.pin2 = val2;
26     Aboard.pin3 = val3;
27     Aboard.pin4 = val4;
28     Aboard.pin5 = val5;
29     Aboard.pin6 = val6;
30     Aboard.pin7 = val7;
31     Aboard.pin8 = val8; //End of duplicate section #1
32 }
33
34
```

コピー & ペースト

```
35 void resetAllPins() {
36     int val1 = 0, val2 = 1, val3 = 0,
37     val4 = 0, val5 = 1, val6 = 0,
38     val7 = 0, val8 = 1;
39     Aboard.pin1 = val1; //Beginning of duplicate section #2
40     Aboard.pin2 = val2;
41     Aboard.pin3 = val3;
42     Aboard.pin4 = val4;
43     Aboard.pin5 = val5;
44     Aboard.pin6 = val6;
45     Aboard.pin7 = val7;
46     Aboard.pin8 = val8; //End of duplicate section #2
47 }
```

Polyspace Code Prover でコードの安全性を証明する 形式手法による解析 – No False Positive

ランタイムエラーの有無
を証明する

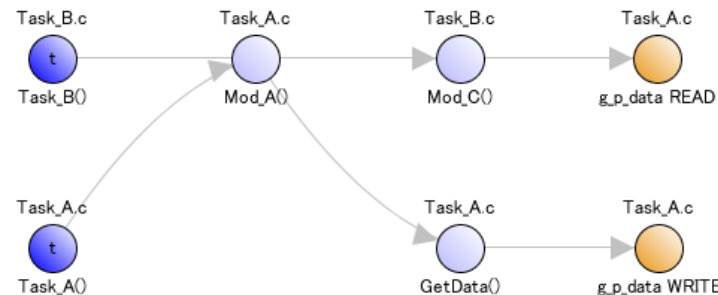


ランタイムエラー

- ゼロ除算
- 整数オーバーフロー
- 範囲外の配列インデックス
- 不適切にデリファレンスされたポインター
- 無限ループ
- 到達不能コード (デッドコード)
- AUTOSAR関連

など、証明可能

複数タスク間でのグローバル
変数の競合を特定する



グローバル変数

- マルチタスク間で使用され、同時アクセスから保護されたグローバル変数
- マルチタスク間で使用され、同時アクセスから保護されていないグローバル変数
- シングルタスクで使用されるグローバル変数
- 宣言されたが未使用のグローバル変数

スタック使用量を
測定する



プロジェクトメトリクス

- 最大スタック使用量のプログラミング
- 最小スタック使用量のプログラミング

関数メトリクス

- 最大スタック使用量
- 最小スタック使用量

コーディングガイドラインのチェックだけでは十分ではない理由

MISRA C:2012準拠 != 脆弱性なし

MISRA C:2012に準拠していてもバグは発見される場合がある

解決策

- Polyspace Bug Finder のバグチェッカーを使用してバグを検出する

? 標準ライブラリ ルーチンの無効な使用
Warning: function 'memmove' is called with possibly invalid argument(s)

- Checks on first argument (destination):
 - ✓ Not null. **第3引数の指定は割り当てられたメモリを超える可能性があります**
 - ? May not be a memory area that is accessible within the boundary given by the third argument.
Actual value of first argument (pointer to void): points at offset 20 in buffer of [1 .. 20] bytes.
 - Actual value of third argument (unsigned int 32): full-range [0 .. 2³²-1]
- Checks on second argument (source):
 - ✓ Not null.
 - ✓ Is a memory area that is accessible within the boundary given by the third argument.
Actual value of second argument (pointer to const void): points at offset multiple of 4 in [24 .. 60]
 - Actual value of third argument (unsigned int 32): full-range [0 .. 2³²-1]

```

Source Code
FreeRTOS_DHCP.c x FreeRTOS_ARP.c x FreeRTOS_IP.c x aws_secure_sockets.c x
1526
1527     memmove( pucTarget, pucSource, xMoveLen );
1528     pxNetworkBuffer->xDataLength -= optlen;
1529
  
```

memmoveの引数に矛盾がある → DoS!
CERTやMISRAによってチェックされない...

MISRA C:2012違反 != 脆弱性あり

MISRA C:2012に逸脱していても安全な場合がある

解決策

- Polyspace Code Prover で解析して安全性を証明する

Result Details

Result Review

Severity: Not a defect | Proven correct

Status: No action planned

Select one or more results to review:

- ✓* **Overflow**
- ✓* **MISRA C:2012 10.4 (Required)**

▼ MISRA C:2012 10.4 (Required) ?

Both operands of an operator in which the usual arithmetic conversions are performed. The left operand of the + operator has essentially signed type while the right operand has essentially unsigned type.

✓* **Overflow** ?

Operation [+] on scalar does not overflow in INT32 range
operator + on type int 32
left: 1
right: [0 .. 127] or [65408 .. 65535]
result: [1 .. 128] or [65409 .. 65536]
(result is truncated)

```

75
76 /*****
77 Conversion from unsigned to signed
78 *****/
79
80 s32b = 1u;
81 s16a = u8a;
82 s32a = s32b + u16a;
83 use_int32 ( u16a );
84
85 /*****
86 Conversion from integer
87 *****/
88
89 f32a = s16a;
90 f32b = 42;
91 f32c = 51u;
92 use_float32 ( s32a );
  
```

MISRA Violation

No loss of sign or value

異なるデータ型での演算 → 問題なし! *上記サンプルコードの場合
MISRA違反を無視/正当化しても安全

入力データとユーザー関数/スタブの戻り値に制約を与える 設計に基づいてより精度の高い解析をする

初期値を設定した場合 (設計に基づいた解析)

名前	ファイル	属性	データ型	main ジェネレータ...	初期化モード	初期化範囲
グローバル変数						
ユーザー定義関数						
Begin_CS()	tasks2.c			MAIN GENERATOR		
Close_To_Zero()	example.c	static		MAIN GENERATOR		
Command_Ordering()	tasks2.c	static		MAIN GENERATOR		
Compute_Injection()	tasks2.c			MAIN GENERATOR		
Computing_from_Sensors()	tasks2.c	static		MAIN GENERATOR		
End_CS()	tasks2.c			MAIN GENERATOR		
Exec_One_Cycle()	tasks2.c			MAIN GENERATOR		
Get_PowerLevel()	tasks2.c			MAIN GENERATOR		
Increase_PowerLevel()	tasks2.c			MAIN GENERATOR		
Meaningless_Function()	example.c			MAIN GENERATOR		
Meaningless_Function.arg1	example.c		int16		INIT	0..10
Meaningless_Function.return	example.c		int16			

引数の初期値を
設定する

```
l_data1 = (int16_T) (arg_In1 * 10);
```

[0 .. 10]

変数レンジの
最小値/最大値

```
/* Outputs for Atomic SubSystem: '<Root>/sub_function' */
```

```
/* Output: '<Root>/out_data' */
```

```
range_propagation_sub_function(l_data1, &arg_Out1);
```

```
/* End of Outputs for SubSystem: '<Root>/sub_function' */
```

```
return arg_Out1;
```

[0 .. 10000]

入力データと戻り値の制約

- グローバル変数
- ユーザー定義関数の引数
- ユーザー定義関数の戻り値
- スタブの戻り値

使用例

- モジュールまたはライブラリの解析
- デバイスドライバ (レジスタアクセス) の解析
- 他モジュールのスタブ化時の解析

初期値を設定しなかった場合 (一般的な静的解析 = 誤検知の可能性)

オーバーフロー発生

```
l_data1 = (int16_T) (arg_In1 * 10);
```

[-32768 .. 32767]

データ型の
最小値/最大値

```
/* Outputs for Atomic SubSystem: '<Root>/sub_function' */
```

```
/* Output: '<Root>/out_data' */
```

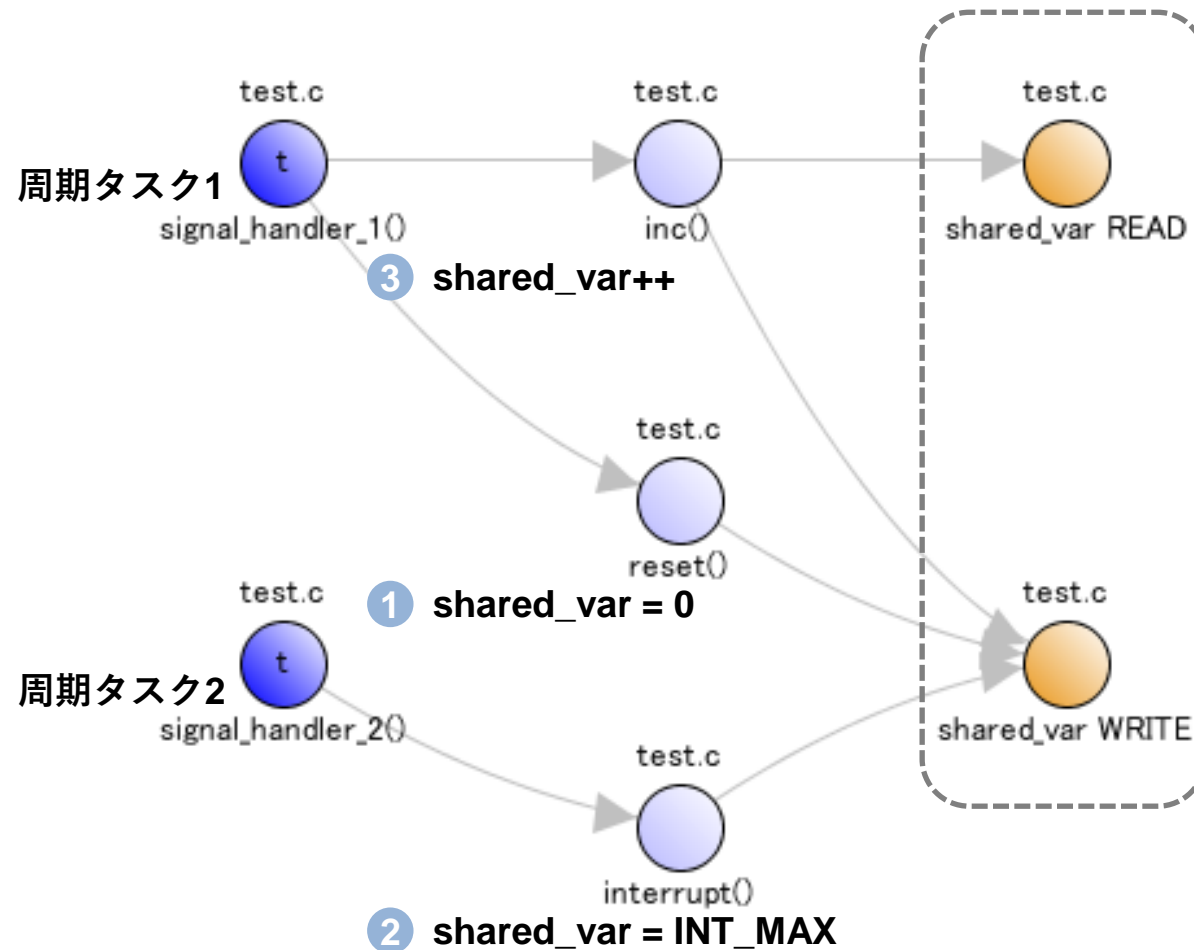
```
range_propagation_sub_function(l_data1, &arg_Out1);
```

```
/* End of Outputs for SubSystem: '<Root>/sub_function' */
```

```
return arg_Out1;
```

マルチタスキング プログラムを解析する

グローバル変数のデータ競合を特定する

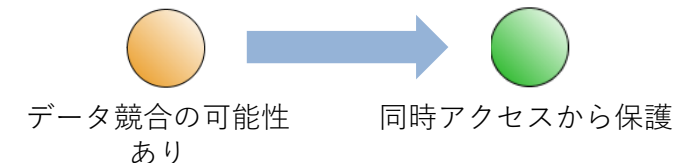


	イベント	ファイル	スコープ	行
1	書き込み値: 0	test.c	<code>_init_globals()</code>	2
2	書き込み値: 1	test.c	<code>inc()</code>	5
3	書き込み値: 0	test.c	<code>reset()</code>	9
	書き込み値: $2^{31}-1$	test.c	<code>interrupt()</code>	13
	読み取り値: 0 or $2^{31}-1$	test.c	<code>inc()</code>	5

データ競合により **オーバーフロー** する可能性

データ競合の解決策

- クリティカルセクションを使用した保護
- 時間的に排他なタスクを使用した保護
- 優先順位を使用した保護
- 割り込みの無効化による保護



プログラムで使用するスタック使用量を見積もる

スタックオーバーフローやメモリの浪費を回避する

```

51 int stub_func(int y);
52
53 int return_code(int y) {
54     int ret;
55     ret = stub_func(y);
56     return (2 * ret);
57 }
58
59 int degree_computation(int a, int b, int c, int x) {
60     int y;
61     int (*funcptr)(int);
62     funcptr = return_code;
63
64     y = a * x * x + b * x + c;
65     if ((y < -93) || (y > 63)) {
66         return funcptr(y);
67     } else {
68         return 2 * x + 1;
69     }
70 }

```

実体のない関数を
自動スタブ化
(デフォルトスタック使用量：0)

関数ポインターによる呼び
出しを考慮
(呼び出し階層参照)

fx 呼び出し階層			
呼び出し先	ファイル	行	スタブ
degree_computation()	initialisations.c	59	
return_code()	initialisations.c	66	
stub_func()	initialisations.c	55	自動

degree_computation関数のスタック使用量

★ 最大スタック使用量 (Value: 40)

★ 最小スタック使用量 (Value: 28)

return_code関数のスタック使用量

★ 最大スタック使用量 (Value: 12)

★ 最小スタック使用量 (Value: 12)

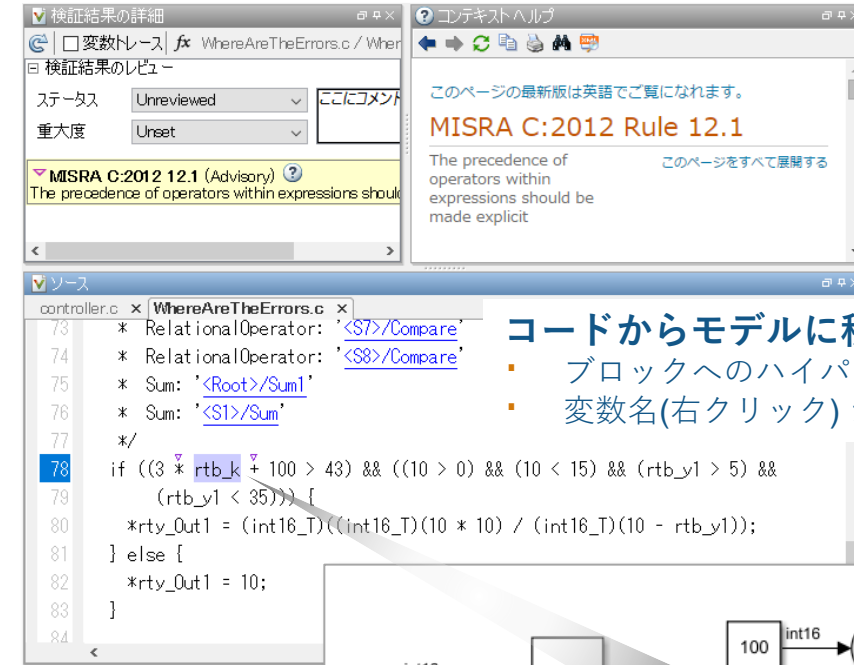
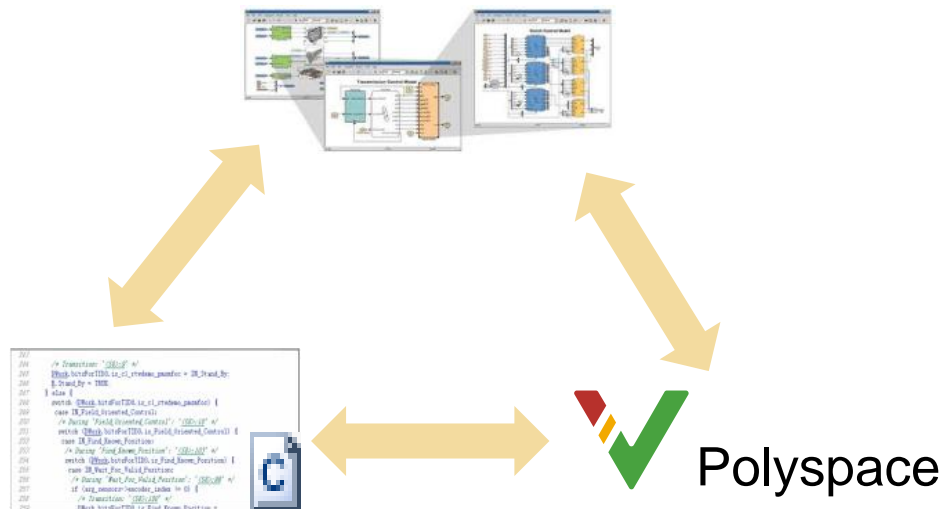
stub_func関数のスタック使用量

- デフォルトスタック使用量：0
- xmlファイルにスタブ関数のスタック使用量を定義して、プログラム全体のスタック使用量の算出に使用可能

* 引数、戻り値、ローカル変数、サブ関数のスタック使用量から算出
 * 本サンプルではint、ポインタを4byteとして算出

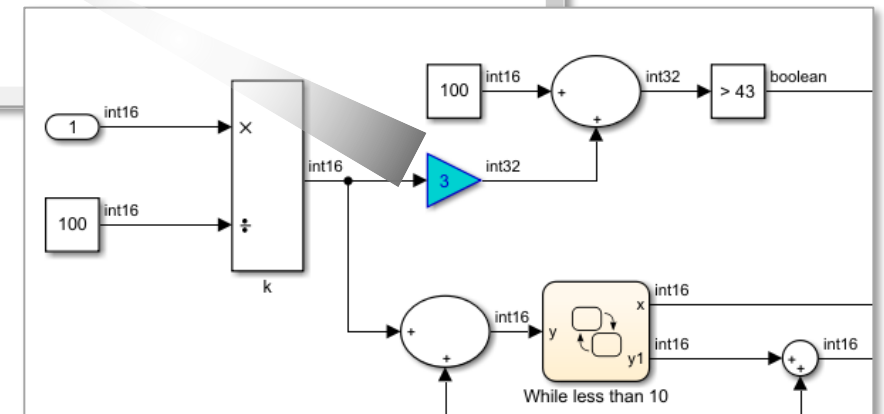
Polyspace による効率化された自動生成コードの検証

- Simulink®環境からPolyspace解析を実行
- Polyspace解析結果を該当ブロックで強調表示
- ブロックにPolyspaceの注釈を追加
- S-Functionコード, カスタムCコードの検証
- MISRA C, CERT C のチェック



コードからモデルに移動

- ブロックへのハイパーリンク選択
- 変数名(右クリック)から選択 **R2017a**



C/C++向け Polyspace 製品

静的解析

Polyspace Bug Finder / Polyspace Bug Finder Server

- コードメトリクスを測定する
- コーディング規約の準拠違反を検出する
- バグ/セキュリティ脆弱性を検出する



静的解析

Polyspace Code Prover / Polyspace Code Prover Server

- ランタイムエラーの有無を証明する
- グローバル変数の競合を特定する
- スタック使用量を測定する

静的解析

Polyspace as You Code *1

- IDE上で単一ファイルの解析を行う
- コーディング規約とバグを検出する
- Polyspace Accessと連携する



R2023b

動的(単体/統合)テスト

Polyspace Test

- テストケースを自動生成する
- テストを実行する (ホスト/ボード)
- テストを管理する



レビュー

Polyspace Access

- Webブラウザ上で結果と品質メトリクスを管理する
- バグの傾向追跡や過去の解析結果と比較する
- バグ管理ツールと連携する (JIRA/Redmine)

*1: IDE機能拡張プラグイン

REVIEW

Dashboard

Current Run ID 7 - Job 1.0 - Uplo...

Run-time Checks

Defects

Coding Standards

Code Metrics

Global Variables

To Do

In Progress

Done

Show only Comment, filename, etc.

Filter out Comment, filename, etc.

Layout

Open in Desktop

Polyspace Access

Showing: 33 / 490

Run-time Checks

AND

In Progress OR To Do

PROJECT EXPLORER

PROJECT DETAILS

Project

Name _debug_C

Language C

Tools Code Prover

Coding Standards MISRA C:2012

Number of Runs 1

Current Run (ID 76)

Upload Date 6/5/20, 10:51 AM

Job 1.0

FILE EXPLORER

example.c

initialisations.c

main.c

single_file_analysis.c

tasks1.c

tasks2.c

Family

ID

Type

Group

Check

Information

Detail

619955	Orange Check	Numerical	Division by zero		Warning: float division b...
619923	Red Check	Static memory	Illegally dereferenced p...		Error: pointer is outside ...
619928	Orange Check	Static memory	Illegally dereferenced p...	Origin: Possibly impacte...	Warning: pointer may b...
620064	Orange Check	Static memory	Illegally dereferenced p...	Origin: Path related issue	Warning: pointer may b...
619991	Red Check	Other	Invalid use of standard li...		Error: function 'sqrt' is c...
619987	Orange Check	Data flow	Non-initialized local vari...	Origin: Possibly impacte...	Warning: local variable ...
620014	Orange Check	Data flow	Non-initialized local vari...	Origin: Path related issue	Warning: local variable ...
620062	Orange Check	Data flow	Non-initialized local vari...	Origin: Possibly impacte...	Warning: local variable ...
620063	Orange Check	Data flow	Non-initialized local vari...	Origin: Possibly impacte...	Warning: local variable ...
620095	Orange Check	Data flow	Non-initialized local vari...	Origin: Possibly impacte...	Warning: local variable ...
620166	Orange Check	Data flow	Non-initialized local vari...		Warning: local variable ...
620173	Orange Check	Data flow	Non-initialized local vari...	Origin: Possibly impacte...	Warning: local variable ...
620233	Red Check	Control flow	Non-terminating call		The called function exa...
620074	Red Check	Control flow	Non-terminating loop		The loop is infinite or co...
620089	Red Check	Static memory	Out of bounds array index		Error: array index is out...
620161	Orange Check	Static memory	Out of bounds array index	Origin: Possibly impacte...	Warning: array index m...
619967	Orange Check	Numerical	Overflow	Origin: Possibly impacte...	Warning: operation [-] o...
619970	Orange Check	Numerical	Overflow	Origin: Possibly impacte...	Warning: operation [+] o...

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer is NULL or points outside its bounds.

The check message shows you the pointer offset and buffer size in bytes. A pointer points outside its bounds when the sum of the offset and pointer size exceeds the buffer size.

- Buffer:** When you assign an address to a pointer, a block of memory is allocated to the pointer. You cannot access memory beyond that block using the pointer. The size of this block is the buffer size.
- Sometimes, instead of a definite value, the size can be a range. For instance, if you create a buffer dynamically using malloc with an unknown input for the size, Polyspace® assumes that the array size can take the full range of values allowed by the input data type.
- Offset:** You can move a pointer within the allowed memory block by using pointer arithmetic. The difference between the initial location of the pointer and its current location is the offset.
- Sometimes, instead of a definite value, the offset can be a range. For instance, if you access an array in a loop, the offset changes value in each loop iteration and takes a range of values throughout the loop.

For instance, if the pointer points to an array:

- The buffer size is the array size.
- The offset is the difference between the beginning of the array and the current location of the pointer.

Diagnosing This Check

Review and Fix Illegally Dereferenced Pointer Checks

Examples

Pointer points outside array bounds

Event

File

Scope

1	Entering function 'interpolation'	main.c	main()
2	Iterating on loop: loop ran 9 times	main.c	interpolation()
3	Illegally dereferenced pointer	main.c	interpolation()

Error Call Graph

main.c

main()

main.c

interpolation()

main.c

IDP

Source Code

example.c

single_file_analysis.c

initialisations.c

main.c

main.c

for (i = 0; i < MAX_SIZE; i++) {

arr++;

if ((found == false) && (*arr > 16)) {

if ((found == 0) && (*arr > 16)) {

found = true;

item = i;

return item;

void main(void)

{

int temp;

PowerLevel = -10000;

found = true;

operator > on type int 32

left: 0 or 12

right: 16

検出の詳細とレビュー

イベントツリー

ソースコード

変数の値確認

解析結果

コンテキストヘルプ

欠陥とコーディングルール違反をワンクリックで修正する 推奨されるコードをすばやく適用する

```
1  #include <iostream>
2  #include <string>
3  #include <cstdlib>
4
5  #define MAX_CNT 7
6  void bug_unusedparameter(int32_t i, int32_t j);
7  void print_int(int32_t i);
8  int32_t main()
9  {
10     int32_t cnt;
11     int32_t i = 0;
12     for (cnt = 0; cnt < MAX_CNT; cnt++)
13     {
14         std::cout << "my test" << std::endl;
15     }
16     std::cout << "my test" << std::endl;
17
18     bug_unusedparameter(i, cnt);
19 }
20
21 void bug_unusedparameter(int32_t i, int32_t j)
22 {
23     print_int(i);
24 }
```

欠陥チェック

無意味なインクルード

インクルードを削除

欠陥チェック

未変更の変数に const 修飾子が付いていません

const 修飾子を追加

欠陥チェック

std::endl によって不要なフラッシュが生じることがあります

ワンクリック
修正

“\n” または
“\n” << std::flush に
置換

欠陥チェック

未使用のパラメーターです

未使用パラメータをコ
メントアウト

*C言語の場合

MISRA C:2012 Rule 2.7

Polyspace Test による組込みシステムでのC/C++コードのテスト開発、管理、実行

- コードテストの自動化と簡素化
 - 静的解析と動的テストの統合
 - コードに対する情報の分析
 - モックとスタブの生成
- 面倒な作業を排除
 - コードテストの強化
 - テストケースの自動生成
 - 認証のためのコードカバレッジの測定



Polyspace を活用したコード検証ワークフロー



Desktop

GUI
コマンドライン/I/F

Polyspace Code
Prover

Polyspace Bug
Finder



Software
Developer



Software
Developer



Polyspace as You
Code

R2021a



Simulink

Embedded Coder

MBD Developer

Desktop ワークフロー



ツール担当者



Server



解析の自動実行

Source code
Repo

コマンドライン/I/F

Polyspace Code
Prover Server

Polyspace Bug
Finder Server



On-Premise

or



Cloud R2021b

Continuous Integration ワークフロー



マネージャー/
チームリーダー



レビューワー



Web Browser

オンライン
レビュー

Jira

R2019a

REDMINE R2020a

flexible project management

Webサーバー



Polyspace
Access

解析結果
アップロード

TÜV SÜD社より各機能安全規格のツール認証を取得済み ～ IEC Certification Kit ～

- コーディング規約
 - MISRA-C
 - MISRA-C++
 - JSF++
- ソフトウェアメトリクス
 - HIS Source Code Metrics
 - <http://www.automotive-his.de/>
- 認証規格
 - IEC 61508, ISO 26262, EN 50128, EN 50657, IEC 62304, ISO 25119
 - IEC Certification Kit

ATE ◆ 認証証書 ◆ CERTIFICATE ◆ CERTIFICADO ◆ CERTIFICAT



CERTIFICATE

No. Z10 067052 0018 Rev. 05

Holder of Certificate: The MathWorks, Inc.
3 Apple Hill Drive
Natick MA 01760-2098
USA

Certification Mark:



Product: Software Tool for Safety Related Development

Model(s): Polyspace® Code Prover™
Polyspace® Code Prover™ Server™
Polyspace® Bug Finder™
Polyspace® Bug Finder™ Server™
Polyspace® Client™ and
Polyspace® Server™ for C/C++
Polyspace® Test™

Parameters: The verification tools, classified T2, are suitable for use in safety-related development according to IEC 61508, EN 50128 and EN 50657 for any SIL and ISO 25119 for any SRL. The verification tools are qualified tools according to ISO 26262 for any ASIL. They are suitably validated for use in safety-related development according to IEC 62304. The report MN74651C is a mandatory part of this certificate.

Tested according to: IEC 61508-3:2010
IEC 62304:2015
ISO 25119-3:2018/AMD1:2020
ISO 26262-8:2018
EN 50128:2011/A2:2020
EN 50657:2017

Polyspace の学習方法について

Polyspace チュートリアル (無償)

Polyspace ハンズオン・セミナー (無償)

Polyspace トレーニング (有償)

Polyspace Bug Finder 入門

静的解析によるソフトウェア バグの特定

R2022a

Polyspace® Bug Finder™ は、C および C++ 組み込みソフトウェアにおけるランタイム エラー、同時実行の問題、セキュリティの脆弱性などの欠陥を特定します。Polyspace Bug Finder では、セマンティクス解析などの静的解析を使用して、ソフトウェア制御、データ フロー、および手続き間の動作を解析します。欠陥を検出して直ちに強調表示することにより、開発プロセスの初期段階でバグを重大度により順位付けし、修正を行うことができます。

Polyspace Bug Finder は、MISRA C®, MISRA C++, JSF++, CERT® C, CERT C++, カスタム命名規則などのコーディング ルール規約への準拠をチェックします。検出したバグやコード ルール違反、および循環的複雑度などのコード品質メトリクスから構成されるレポートを生成します。Polyspace Bug Finder は、Eclipse™ IDE と共にデスクトップでのコード解析に使用できます。

自動生成されたコードについては、Polyspace の検証結果を Simulink® モデルおよび dSPACE® TargetLink® ブロックまで遡って追跡できます。

業界標準には、IEC Certification Kit (for ISO 26262 and IEC 61508) と DO Qualification Kit (for DO-178) によって対応しています。

ソフトウェアバグを根絶するC/C++静的コード解析 Polyspace ハンズオン・セミナー

<概要>
近年のソフトウェアは、高度な機能を実現する為に、複雑化・大規模化する傾向に加え、信頼性や安全性に対する要求も高まっています。本セミナーでは、静的コード解析ツール“Polyspace Bug Finder”と“Polyspace Code Prover”を用いて、ソフトウェアの欠陥検出、セキュリティ脆弱性の検出、ランタイムエラーが含まれない事の証明方法をご紹介します。

	時間	アジェンダ
第1部	30分	多用途の静的コード解析手法を提供するPolyspace紹介 (講義式)
第2部	1時間30分	Polyspaceを使用した静的コード解析 (ハンズオン形式) *休憩10分含む
第3部	30分	QA会 (講義式)

*ご興味のあるセッションのみにご参加いただけます。

第2部: Polyspaceを使用した静的コード解析 (ハンズオン形式)参加の方へ
ご参加の方は、Polyspace評価版をインストールしたノートPC (Windows64bit) をご持参ください。セミナーにご参加ください。

Polyspace の有効活用に向けて 短期間で習得していただけるような教育カリキュラムをご提供します

PolyspaceによるC/C++コード検証(2日間)

第1日

- Polyspace のワークフローの概要
- Polyspace Bug Finder™ 解析
- Polyspace Code Prover™ 検証結果の解析

第2日

- Polyspace Code Prover™ 検証と結果の管理
- Polyspace Code Prover™ 検証への精度の追加
- 統合解析
- ソフトウェア品質の測定とレポート
- アプリケーション解析



- 自己学習形式
- 主な対象者
 - これから使う方
- 学習ゴール
 - 使用イメージを理解する
- 詳細
 - [デスクトップでの Polyspace Bug Finder の実行](#)
 - [デスクトップでの Polyspace Code Prover の実行](#)

- ハンズオン形式
- 主な対象者
 - これから使う方、使い始めた方
- 学習ゴール
 - 概要を把握する
 - 操作性を体験する
- 詳細
 - 担当営業、または、Polyspace専任営業にお問い合わせください。

- 専任講師によるトレーニング
- 主な対象者
 - これから使う方、使い始めた方
 - ユーザー
- 学習ゴール
 - 一通りの機能を体系的に学ぶ
- 詳細
 - [Polyspace による C/C++ コードの検証](#)

まとめ

Polyspace とは・・・

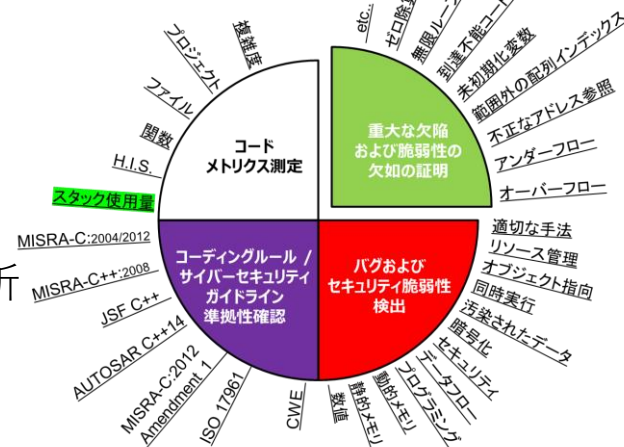
組込みソフトウェア開発のためのC/C++向けの静的コード解析ツール

Polyspace Bug Finder

- 素早くコードを解析

Polyspace Code Prover

- 徹底的にコードを解析



- 機能安全対応
- 自動車・医療・航空



その他の静的コード解析ツールとの違いは・・・

ソフトウェアにランタイムエラーが存在しないことを証明

Simulinkと連携して
自動生成コードの解析
/ レビュー

