次世代システムに求められるソフトウェア検証技術
～静的解析の価値と有効性～

MathWorks Japan
Application Engineering
Application Engineer
能戸 フレッド・Fred Noto

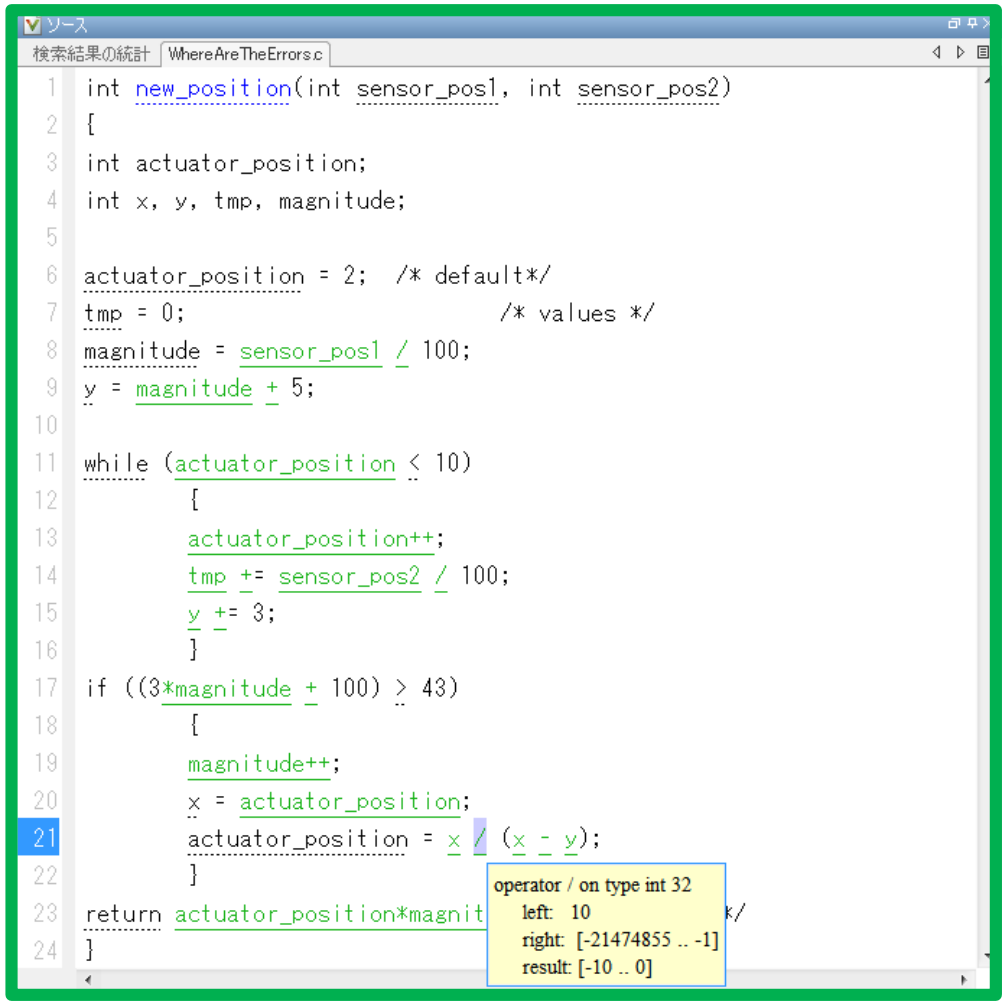
© 2017 The MathWorks, Inc.

Polyspace®のコード証明

Polyspaceは品質保証の上、検証工数の削減に貢献します

- コード内のエラーをどう検出しますか？

- Polyspaceはコードの安全性を保証するための静的解析を提供します
- コードの品質を証明することにより、検証プロセスの効率化に繋がります



アジェンダ

次世代システムの検証に関する課題
～動的テストと静的解析の目的～

- Polyspace静的解析ソリューションの価値と有効性
- ユーザ事例　＆　まとめ

次世代システムのトレンド

IoT　　自律システム

スマートシステム　　等々. . .

新機能　　高機能　　市場への早期参入

高性能　　品質向上　　等々. . .

コンシューマー

次世代システムのトレンド

IoT

ス

新機能

高性能

品質向上

多数のチャレンジ！

セキュリティ脆弱性

システムの複雑化

ソフトウェアの大規模化

スケジュールの最小化

品質の確保

コンシューマー

複雑・高品質ソフトウェアをより早く開発！

効率的な検証手法が必須！

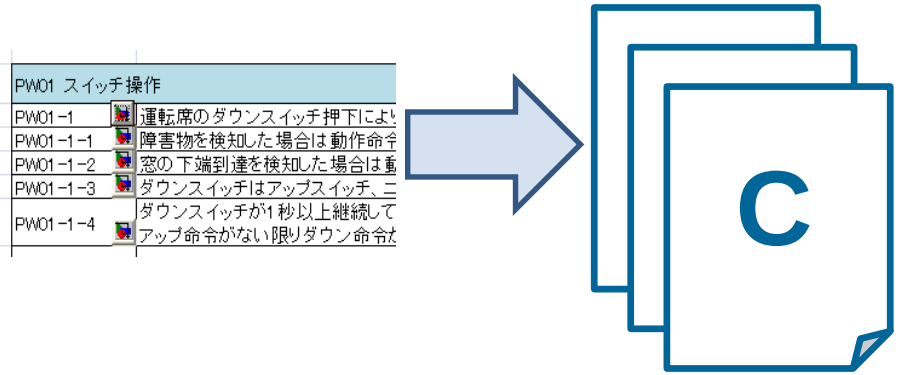
開発者・エンジニア

次世代システムの検証に関する課題

多くの検証目的が存在し、テスト時間が膨張する

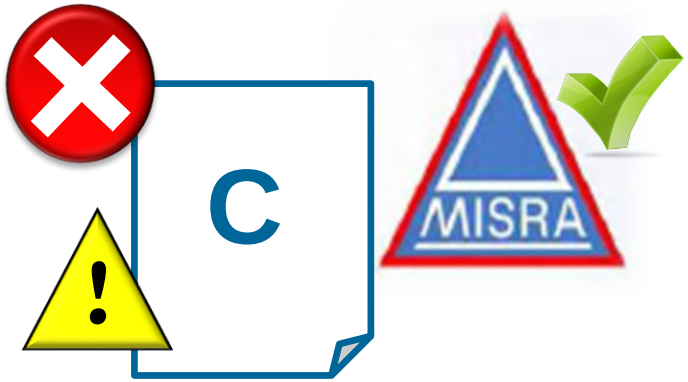
動的テスト

- 要求仕様通りの設計かを確認
- 要求仕様に問題がないことを確認



機能の検証

- エラー・欠陥の存在を確認
- スタンダード準拠の確認

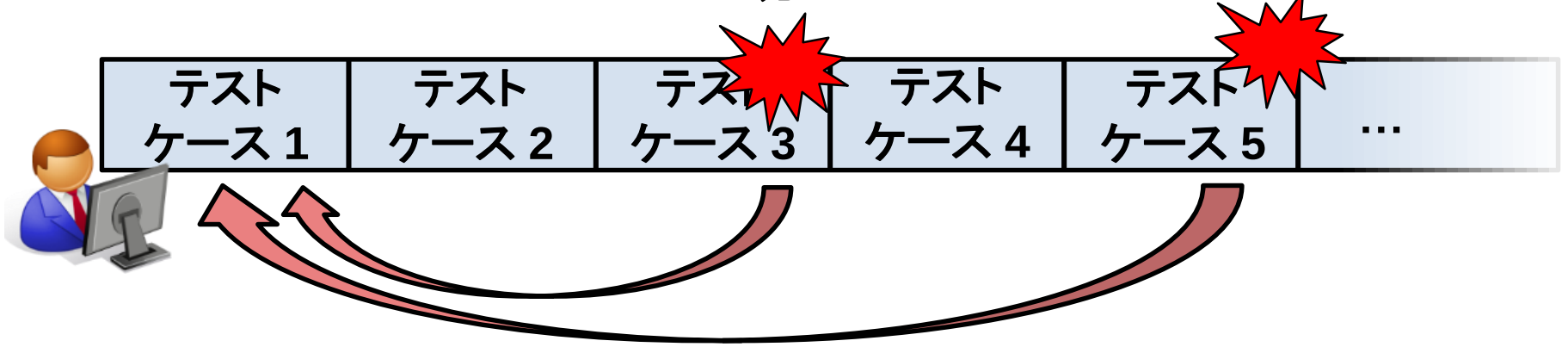


ソフトの検証

動的テスト実行に関する課題

動的テストは動作確認で必要だが、エラー検証には非効率

従来のソフト検証手法：動的テスト



ユニットテスト
&
統合テスト

問題点： エラー検出の遅れ　納期の延期　納品物でエラーが発生
リワーク　多数の繰り返し作業

複雑系の故障

- Ariane-5, ロケット
  - エラータイプ ： オーバーフロー
  - 結果 ： ロケットの破壊

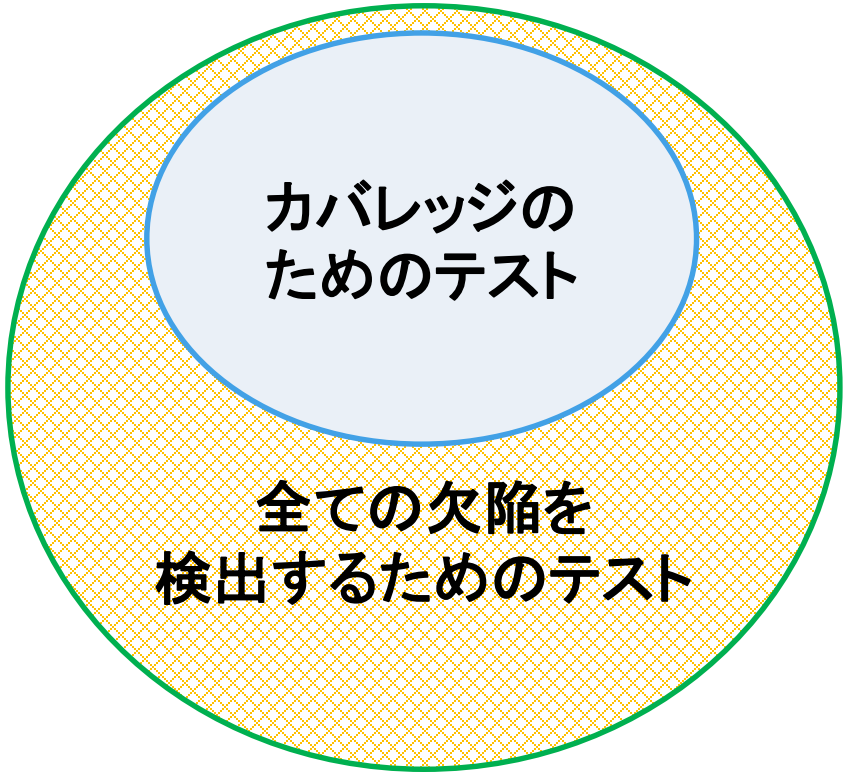
- USS Yorktown,タイコンデロガ級ミサイル巡洋艦
  - エラータイプ ： ゼロ割り
  - 結果 ： 巡洋艦の推進システムが故障

- Therac-25, 放射線療法機器
  - エラータイプ ： オーバーフローと競合状態
  - 結果 ： 数人の患者を死亡させた

ソフトウェア品質確保の効率化に向けて

提案：静的解析によるソフトウェア品質の確保

- 従来的なテスト手法 → テストケースの作成、実行、確認が必要




```
x / (x - y)
```

- 未初期化変数
- オーバーフロー
- ゼロ除算

int32 の場合、
4.61×10^{18} の
テストケースが可能

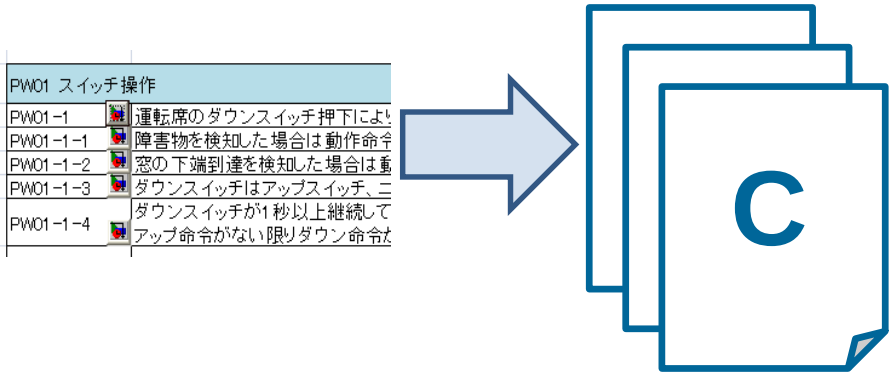
**全て可能なテストケースの
作成・実行・レビューは不可能！**

ソフトウェア検証のストリームライン

検証目的により適切な検証手法を使用してプロセスを効率化

動的テスト

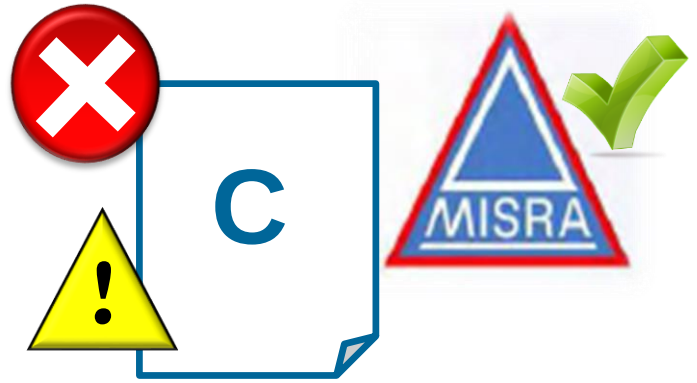
- 要求仕様通りの設計かを確認
- 要求仕様に問題がないことを確認



機能チェック

静的検証

- エラー・欠陥の存在を確認
- スタンダード準拠の確認



ソフト品質

静的解析の種類について

Polyspaceはコード証明を含むソフト品質確保のソリューション

静的解析手法の種類:

シンタックス解析　　モデル検査　　制御フロー解析

セマンティックス解析　　データフロー解析　　形式手法

手法の目的:

コーディング規約への準拠　　コードメトリクス測定

欠陥の検出　　コード証明

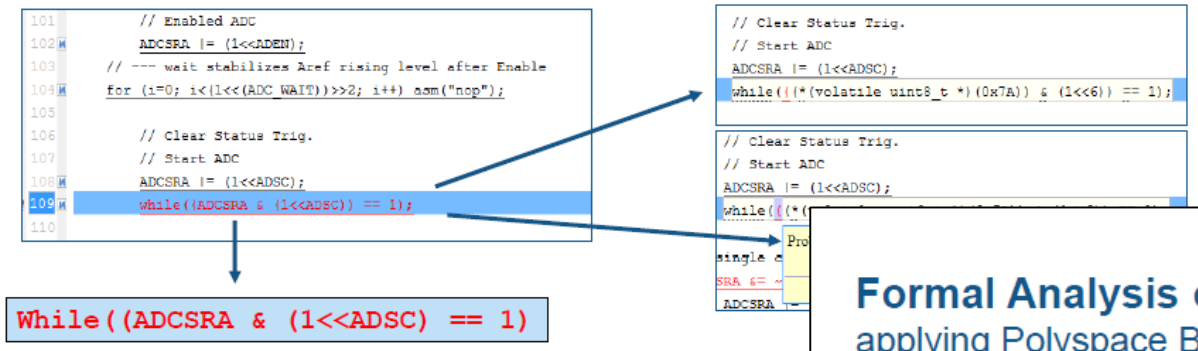
Formal Analysis of Avionic Software to DO-178B

applying Polyspace Bug Finder and Code Prover

SOLARIMPULSE
AROUND THE WORLD IN A SOLAR AIRPLANE

– Latent bug or defect hunting, e.g. incorrect temperature in throttle box

– No test cases or compilation needed



https://www.mathworks.com/company/events/conferences/matlab-conference-switzerland/2015/proceedings/solar-impulse-first-round-the-world-solar-flight-attempt.pdf

Formal Analysis of Avionic Software to DO-178B

applying Polyspace Bug Finder and Code Prover

SOLARIMPULSE
AROUND THE WORLD IN A SOLAR AIRPLANE

- Independent, systematic code reviews, compliance to MISRA-C
- Complexity results to support DO-178B “simple system” argument for case where we had to “re-engineer” design assurance level equivalence
- Bug Finder and Code Prover provided 1-2 Man-Year savings and automated capability in parallel to development which were not available otherwise

アジェンダ

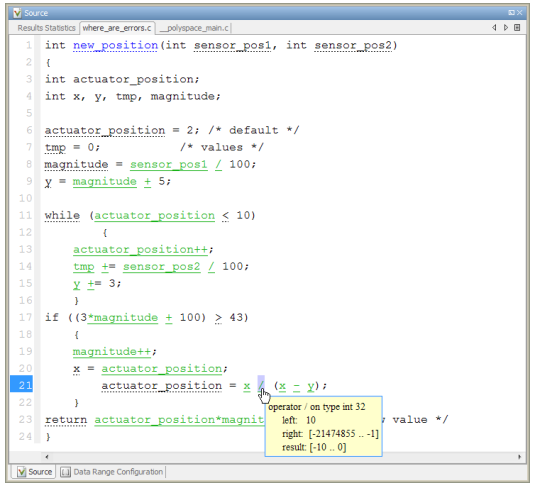
- 次世代システムの検証に関する課題
  ～動的テストと静的解析の目的～

Polyspace静的解析ソリューションの価値と有効性

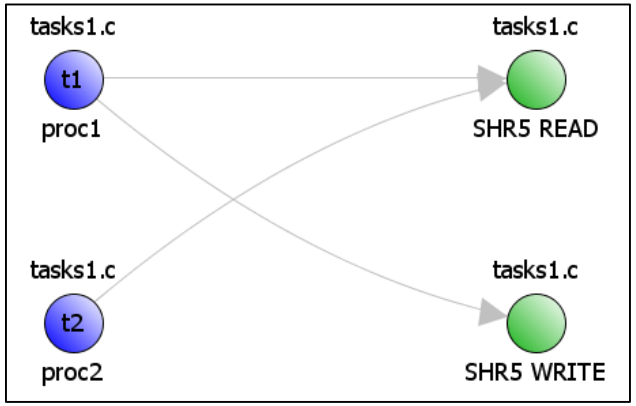
- ユーザ事例　＆　まとめ

Polyspaceによる検証作業の効率化：活用例

早期段階でのコーディング
ルールチェックと、バグや
セキュリティ脆弱性の検出



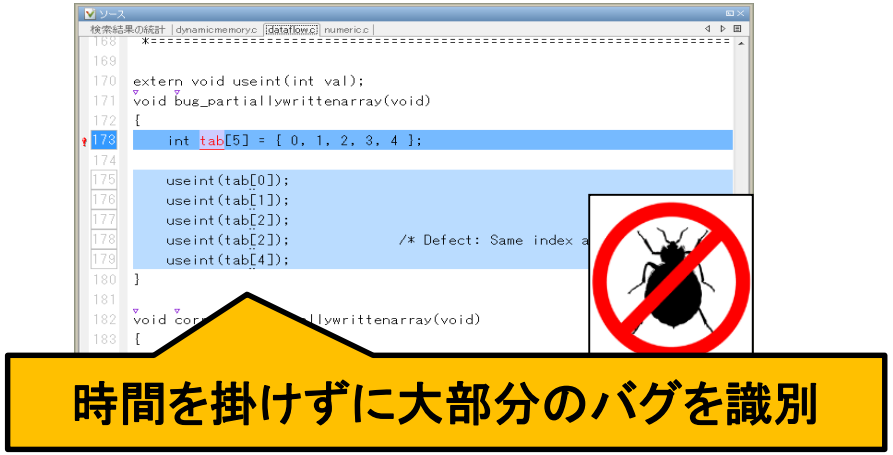
コードレビューの効率化



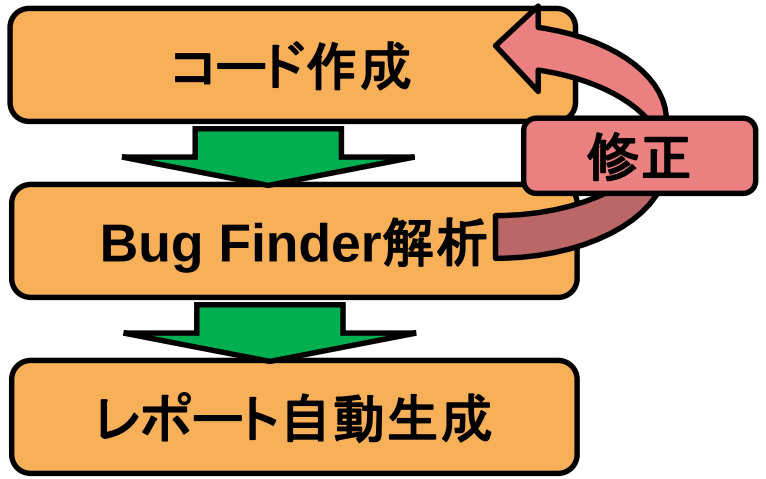
潜在的な構造的
エラーを予防

Polyspace Bug Finder™ のスピーディー解析

- **バグ検出**
  - コード作成後、すぐに欠陥を検出・修正
- **コーディングルールチェック**
  - エラー予防と再利用性向上
  - MISRA準拠を確認
- **コードメトリックス解析**
  - コードの複雑度を測定



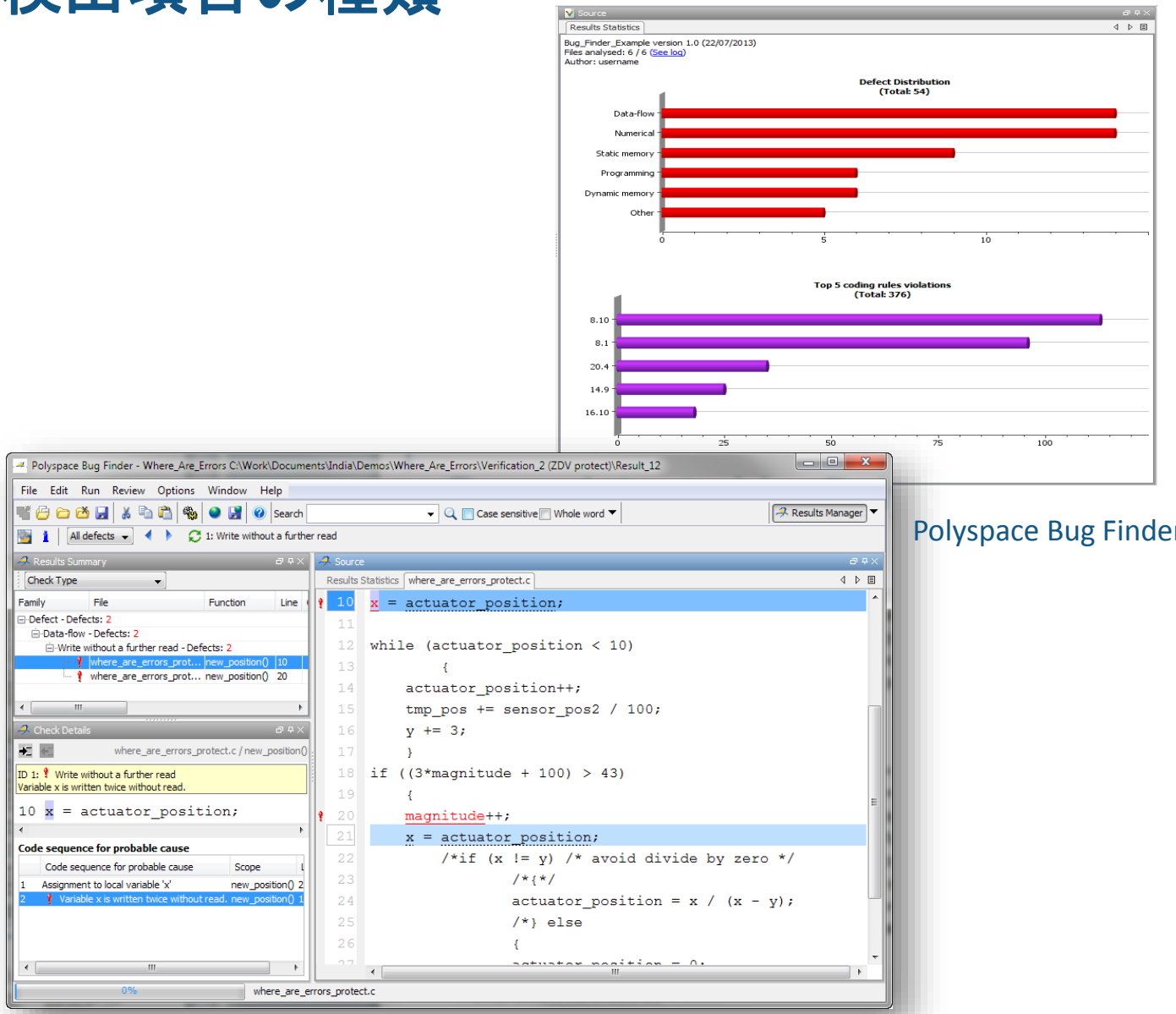
開発プロセスの上流で不具合を発見！
コードを統合前に多くの欠陥を修正！
コード開発の効率に繋がる！



素早い欠陥検出・修正を可能とするPolyspace Bug Finder

Polyspace Bug Finder：検出項目の種類

- 数式エラー
- 静的メモリーエラー
- 動的メモリーエラー
- プログラミングエラー
- データフローエラー
- 同時実行
- セキュリティ
- 汚染されたデータ
- その他



Polyspace Bug Finder GUI

Polyspace Bug Finder：検出項目リスト（1/2）

Numerical

1. Integer division by zero
2. Float division by zero
3. Float conversion overflow
4. Integer conversion overflow
5. Unsigned integer conversion overflow
6. Sign change integer conversion overflow
7. Float overflow
8. Integer overflow
9. Unsigned integer overflow
10. Invalid use of std. library integer routine
11. Invalid use of std. library float routine
12. Shift of a negative value
13. Shift operation overflow

Dataflow

1. Write without further read
2. Non-initialized variable
3. Non-initialized pointer
4. Variable shadowing
5. Missing or invalid return statement
6. Unreachable code
7. Dead code
8. Useless if
9. Partially access array
10. Uncalled function
11. Pointer to non initialized value converted to const pointer
12. Code deactivated by constant false condition

Static memory

1. Array access out of bounds
2. Null pointer
3. Buffer overflow from incorrect string format specifier
4. Destination buffer overflow in string manipulation
5. Destination buffer underflow in string manipulation
6. Pointer access out of bounds
7. Pointer or reference to stack variable leaving scope
8. Unreliable cast of function pointer
9. Unreliable cast of pointer
10. Invalid use of std. library memory routine
11. Invalid us of standard library string routine
12. Arithmetic operation with NULL pointer
13. Use of path manipulation function without maximum sized buffer checking
14. Wrong allocated object size for cast

Programming

1. Assertion
2. Bad file access mode or status
3. Call to memset with unintended value
4. Declaration mismatch
5. Exception caught by value
6. Exception handler hidden by previous handler
7. Format string specifiers and arguments mismatch
8. Improper array initialization
9. Incorrect pointer scaling
10. Invalid assumptions about memory organization
11. Invalid use of == (equality) operator
12. Invalid use of = (assignment) operator
13. Invalid use of floating point operation
14. Invalid use of standard library routine
15. Invalid va_list argument
16. Copy of overlapping memory
17. Missing null in string array
18. Modification of internal buffer returned from nonreentrant standard function
19. Overlapping assignment
20. Possible misuse of sizeof
21. Possibly unintended evaluation of expression because of operator precedence rules
22. Qualifier removed in conversion
23. Standard function call with incorrect arguments
24. Use of memset with size argument zero
25. Wrong type used in sizeof
26. Writing to const qualified object
27. Variable length array with nonpositive size

Polyspace Bug Finder：検出項目リスト（2/2）

Concurrency

1. Data race
2. Data race including atomic operations
3. Deadlock
4. Double lock
5. Double unlock
6. Missing lock
7. Missing unlock

Resource management

1. Closing a previously closed resource
2. Resource leak
3. Use of previously closed resource
4. Writing to read-only resource

Dynamic memory

1. Use of previously freed pointer
2. Unprotected dynamic memory allocation
3. Release of previously deallocated pointer
4. Invalid free of pointer
5. Memory leak
6. Invalid deletion of pointer

Tainted data

1. Array access with tainted index
2. Command executed from externally controlled path
3. Execution of externally controlled command
4. Host change using externally controlled elements
5. Library loaded from externally controlled path
6. Loop bounded with tainted value
7. Memory allocation with tainted size
8. Pointer dereference with tainted offset
9. Tainted division operand
10. Tainted modulo operand
11. Tainted NULL or non-null-terminated string
12. Tainted sign change conversion
13. Tainted size of variable length array
14. Tainted string format
15. Use of externally controlled environment variable
16. Use of tainted pointer

Object oriented

1. *this not returned in copy assignment operator
2. Base class assignment operator not called
3. Base class destructor not virtual
4. Copy constructor not called in initialization list
5. Incompatible types prevent overriding
6. Member not initialized in constructor
7. Missing explicit keyword
8. Missing virtual inheritance
9. Object slicing
10. Partial override of overloaded virtual functions
11. Return of non const handle to encapsulated data member
12. Self assignment not tested in operator

Other

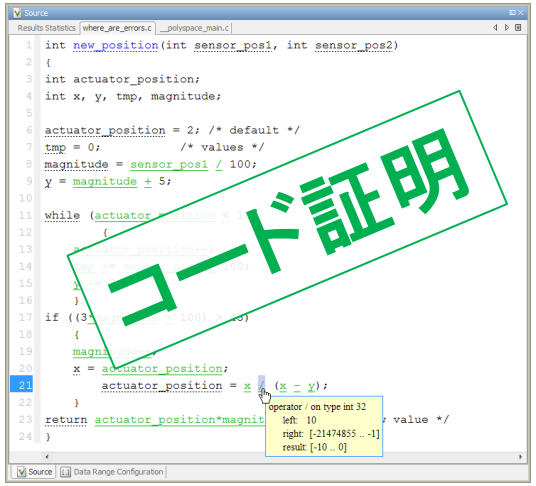
1. Invalid use of other standard library routine
2. Large pass-by-value argument
3. Assertion
4. Format string specifiers and arguments mismatch
5. Line with more than one statement

Security

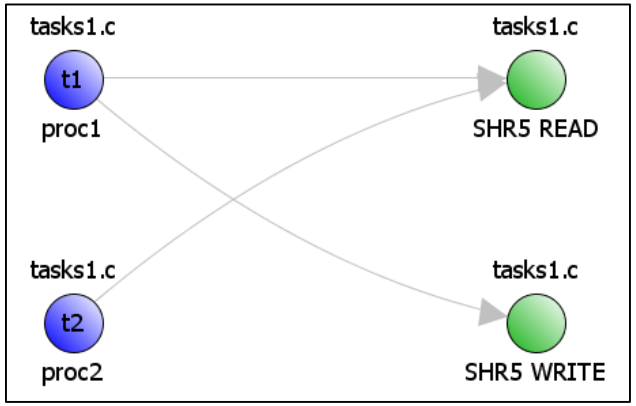
1. File access between time of check and use (TOCTOU)
2. File manipulation after chroot without chdir
3. Use of non-secure temporary file
4. Vulnerable path manipulation
5. Vulnerable permission assignments
6. Function pointer assigned with absolute address
7. Incorrect order of network connection operations
8. Mismatch between data length and size
9. Missing case for switch condition
10. Sensitive data printed out
11. Sensitive heap memory not cleared before release
12. Umask used with chmod-style arguments
13. Uncleared sensitive data in stack
14. Unsafe standard encryption function
15. Unsafe standard function
16. Use of dangerous standard function
17. Use of obsolete standard function
18. Deterministic random output from constant seed
19. Predictable random output from predictable seed
20. Vulnerable pseudo-random number generator
21. Execution of a binary from a relative path can be controlled by an external actor
22. Load of library from a relative path can be controlled by an external actor

Polyspaceによる検証作業の効率化：活用例

早期段階でのコーディングルールチェックと、バグやセキュリティ脆弱性の検出



コードレビューの効率化



潜在的な構造的エラーを予防

コードレビューの課題

コードレビューでの
ランタイムエラー検出は
高コストで全てカバーできない

```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
      *p = 0;
      p++;
    }

  if (get_bus_status() > 0) {
    if (get_oil_pressure() > 0) {
      *p = 5;
    } else {
      i++;
    }
  }

  i = get_bus_status();

  if (i >= 0) {
    *(p - i) = 10;
  }
}
```

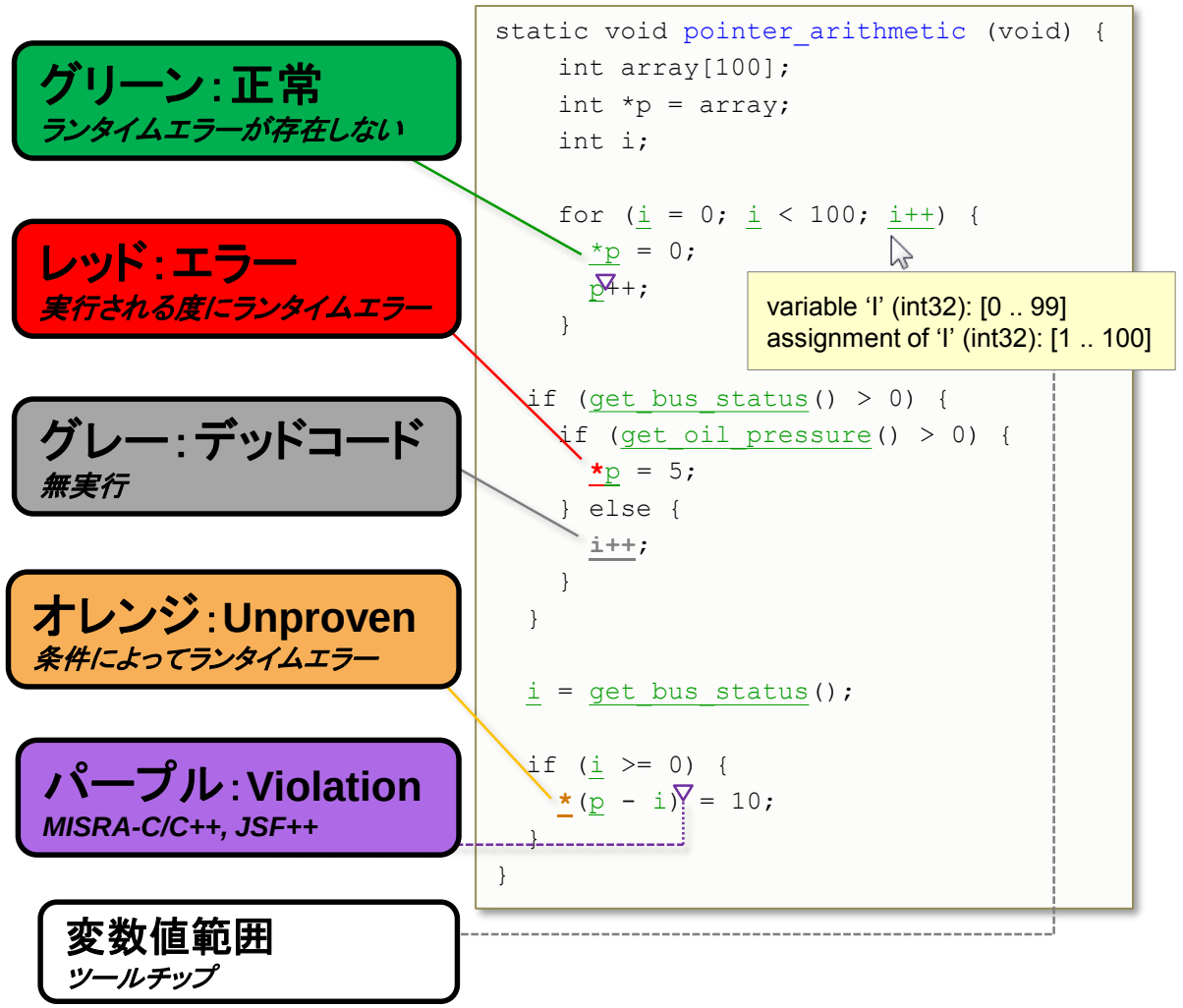
下記のような情報がありましたら如何でしょうか？

Pointer ‘p’ (pointer to int32, size 32bits)
- **Pointer is not null**
- **Points to 4 bytes at offset of 400 in buffer of 400 bytes, so is outside bounds**
- **Pointer may point to ‘array’, local to function ‘pointer_arithmetic’.**

Polyspace Code Prover™でコードの正しさを証明

- **Quality（品質）**
  - ランタイムエラーの証明
  - 測定、向上、管理
- **Usage（使用方法）**
  - コンパイル、プログラム実行、テストケースは不要
  - 対応言語：C/C++/Ada
- **Process（プロセス）**
  - ランタイムエラーの早期検出
  - 自動生成コード、ハンドコードの解析可能
  - コードの信頼性を測定

実機実験前にコード信頼性を確保して手戻りを削減



Polyspaceは全ての実行パスの結果を証明する！

Polyspace Code ProverのKey Point

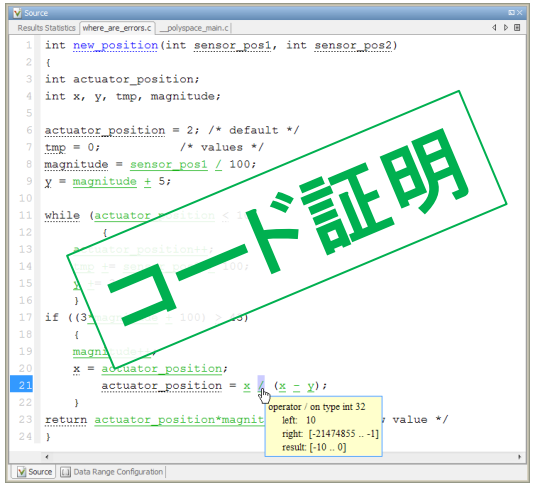
- “実行パス” x “変数レンジ” を検証して
  全てのテストケース実行に同等！

- 抽象解釈（abstract interpretation）により
  “ランタイムエラーフリー” なコードを証明します
  - **コードの正しさの数学的な証明**

不具合の不在をPROVEする事がPolyspaceの特徴

Polyspaceによる検証作業の効率化：活用例

早期段階でのコーディング
ルールチェックと、バグや
セキュリティ脆弱性の検出



コードレビューの効率化

潜在的な構造的
エラーを予防

Polyspaceによる同時実行問題の特定

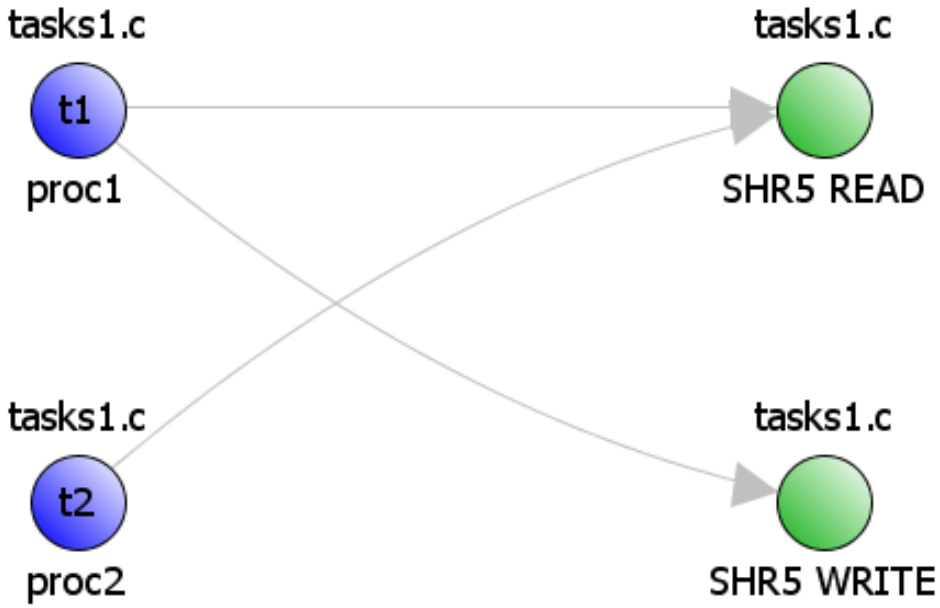
ソフトウェア内のグローバル・共有変数の使用をレビューを加速化

対話的なレビュー手法

```
29  static int SHR5 = 5;
```

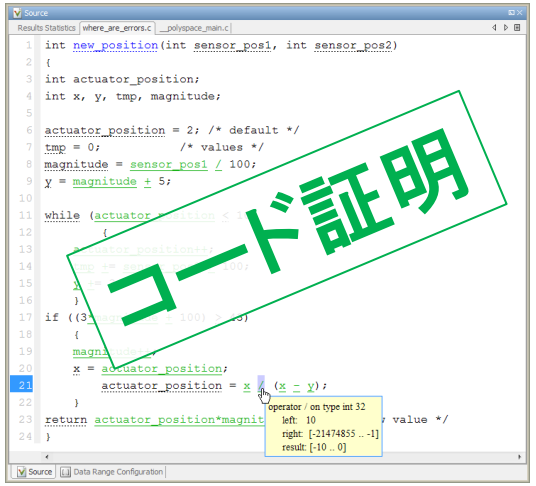
	Event	File	Scope	Line
◀	Written value: 5	tasks1.c	_init_globals()	29
◀	Written value: 28	tasks1.c	proc1()	103
▶	Read value: 5	tasks1.c	proc1()	103
▶	Read value: 5 or 28	tasks1.c	proc2()	112

グラフィカルなレビュー手法

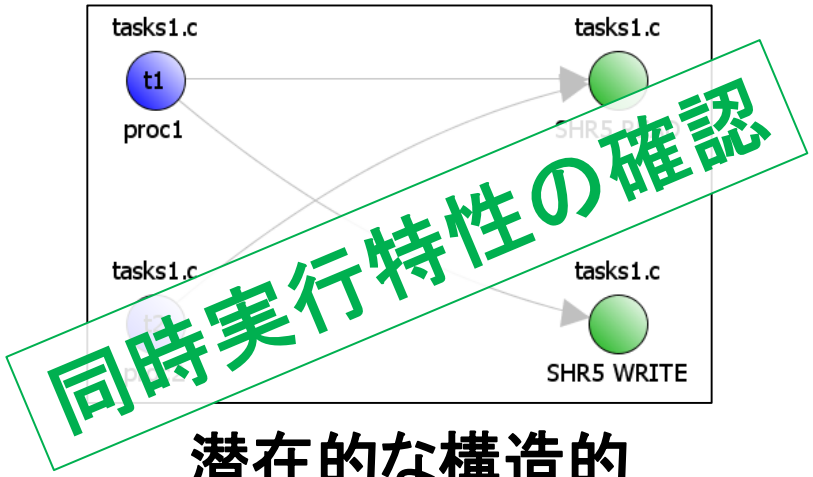


Polyspaceによる検証作業の効率化：活用例

早期段階でのコーディング
ルールチェックと、バグや
セキュリティ脆弱性の検出



コードレビューの効率化



潜在的な構造的
エラーを予防

Polyspaceは下記の検証作業を加速化します

検証プロセスを効率化する静的解析のソリューション

MISRAチェック

コーディングルール準拠

ISO、IEC、DO認証

認証サポート

設計・構造の最適化

コードメトリクス

バグがないソフトを確保

コード証明

コード品質の測定

包括的な静的解析

動的テストの工数を削減

ソフトウェアロバスト性の解析

アジェンダ

- 次世代システムの検証に関する課題
  ～動的テストと静的解析の目的～
- Polyspace静的解析ソリューションの価値と有効性

ユーザ事例　＆　まとめ

Polyspaceユーザー事例

PCCN Controls

ONE FORD
ONE TEAM • ONE PLAN • ONE GOAL

Introducing Polyspace into the Software Development Process

Eileen Davidson
Ford Motor Company

USER STORY

GlucoLight Ensures Reliable Software for Medical Tri
Using PolySpace™ Products for C/C++

Studies show that management of glucose levels reduces infection, length of hospital stay, and mortality for patients in intensive care. Current glycemic control methods are time-consuming and labor-intensive, however, requiring medical staff to draw blood samples and manually measure blood glucose levels.

GlucoLight Corporation is developing a noninvasive, continuous glucose monitoring system that uses imaging technology and requires no manual intervention.

GlucoLight SENTRIS-100, a noninvasive glucose monitoring system.

https://www.mathworks.com/company/events/conferences/automotive-conference-michigan/2015/proceedings/introducing-polyspace-products-into-software-development-process.pdf

MathWorks　MATLAB&SIMULINK

日産自動車

Polyspaceを利用してソフトウェア品質を向上

Nissan Fairlady Z.

課題
ソフトウェア品質の改善に向け、これまで見つけられなかったランタイムエラーを発見すること

ソリューション
MathWorksのツールを利用して、日産やサプライヤが作成したコードを完璧に解析

結果
- サプライヤのコードにバグを見つけることができた
- ソフトウェアの信頼性が改善された
- Polyspace製品が日産のサプライヤに適用できることが分かった

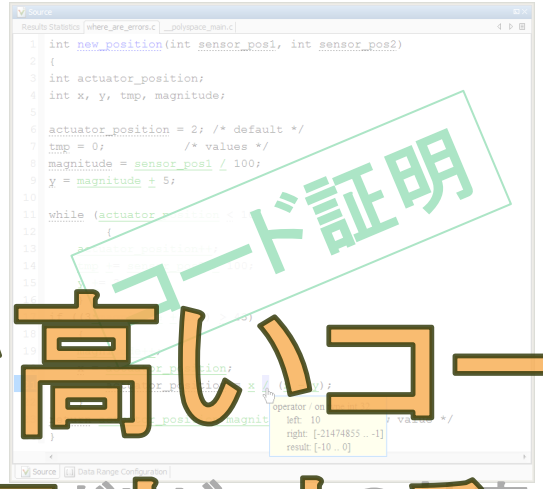
『Polyspace製品を利用して、それまで業界のどのツールもできなかったソフトウェアの信頼性を確保することができました。』

日産自動車　菊池様

1

http://www.mathworks.com/tagteam/77497_91488v01_Nissan_UserStory_final.pdf

Polyspaceをどう活用しますか？



潜在的な構造的
エラーを予防

認証とレポート生成

素早い検証業務と高いコード品質の確保を両立する Polyspace静的解析ソリューション

最後に. . .

Polyspaceの評価にご興味が
ありましたらご連絡ください

（fred.noto@mathworks.co.jp）

Polyspace詳細にご興味が
ありましたら展示ブースにお立寄りください

Thank You For Your Attention

ご清聴ありがとうございました

© 2017 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.