

テスト・デバッグ工数を低減する Polyspace®による静的解析について

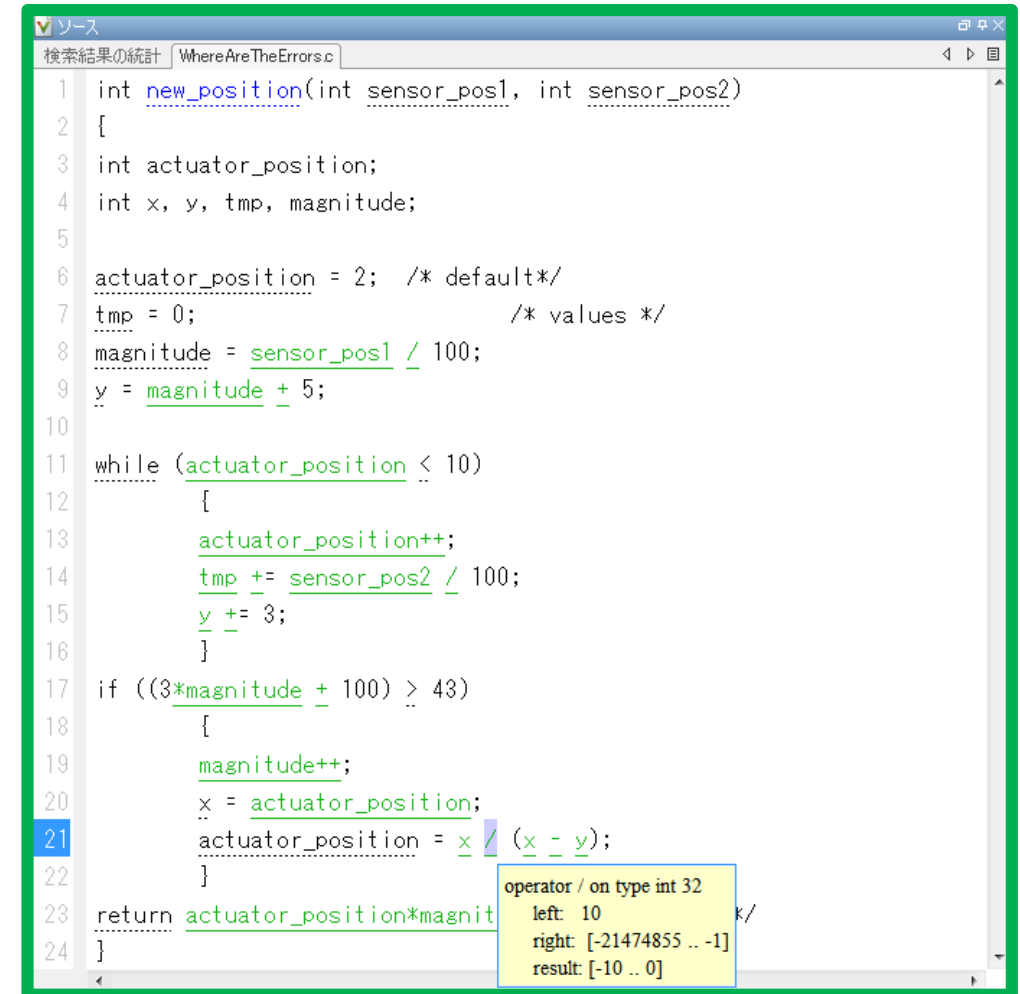
MathWorks Japan
アプリケーションエンジニアリング部
アプリケーションエンジニア
能戸 フレッド ・ Fred Noto
fred.noto@mathworks.co.jp



Polyspaceのコード証明

- コード内にエラーはありますか？

- 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
- テスト実行せずにソフトウェアのロバスト性を確保します



The screenshot shows a MATLAB script editor window titled 'WhereAreTheErrors.c'. The code is as follows:

```
1 int new_position(int sensor_pos1, int sensor_pos2)
2 {
3     int actuator_position;
4     int x, y, tmp, magnitude;
5
6     actuator_position = 2; /* default*/
7     tmp = 0; /* values */
8     magnitude = sensor_pos1 / 100;
9     y = magnitude + 5;
10
11     while (actuator_position < 10)
12     {
13         actuator_position++;
14         tmp += sensor_pos2 / 100;
15         y += 3;
16     }
17     if ((3*magnitude + 100) >= 43)
18     {
19         magnitude++;
20         x = actuator_position;
21         actuator_position = x / (x - y);
22     }
23     return actuator_position*magnitude;
24 }
```

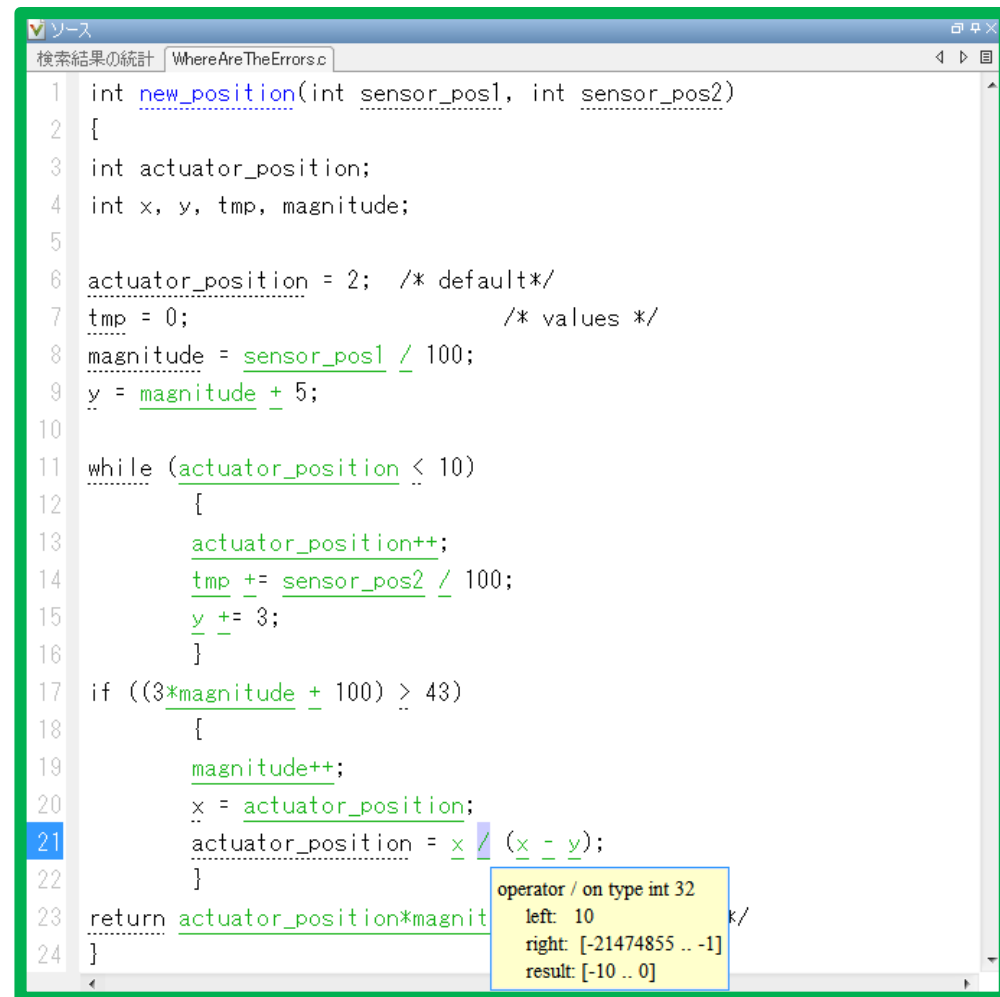
A tooltip is displayed over line 21, indicating a static analysis error:

- operator / on type int 32
- left: 10
- right: [-21474855 .. -1]
- result: [-10 .. 0]

Polyspaceはランタイムエラーの検出だけでなく、
Run-Time Error Free の信頼性を確保します

アジェンダ

- MathWorks・Polyspaceについて
- ソフトウェア検証に関するチャレンジ
- Polyspace静的解析の解決案
- Polyspaceのメリット
- ユーザー事例



```
1 int new_position(int sensor_pos1, int sensor_pos2)
2 {
3     int actuator_position;
4     int x, y, tmp, magnitude;
5
6     actuator_position = 2; /* default*/
7     tmp = 0; /* values */
8     magnitude = sensor_pos1 / 100;
9     y = magnitude + 5;
10
11     while (actuator_position < 10)
12     {
13         actuator_position++;
14         tmp += sensor_pos2 / 100;
15         y += 3;
16     }
17     if ((3*magnitude + 100) >= 43)
18     {
19         magnitude++;
20         x = actuator_position;
21         actuator_position = x / (x - y);
22     }
23     return actuator_position*magnitude;
24 }
```

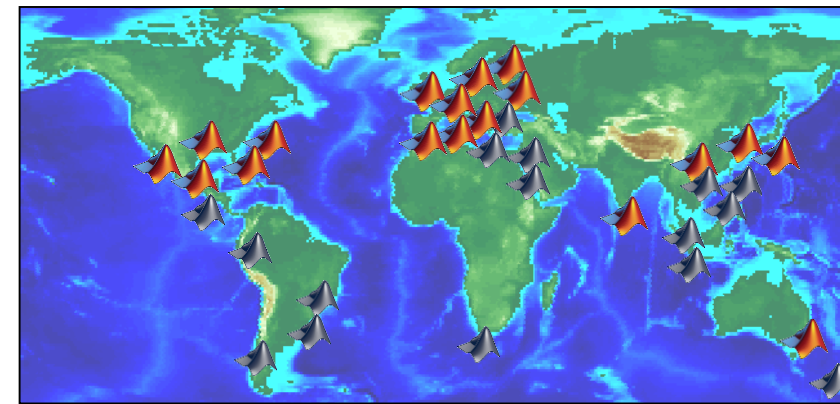
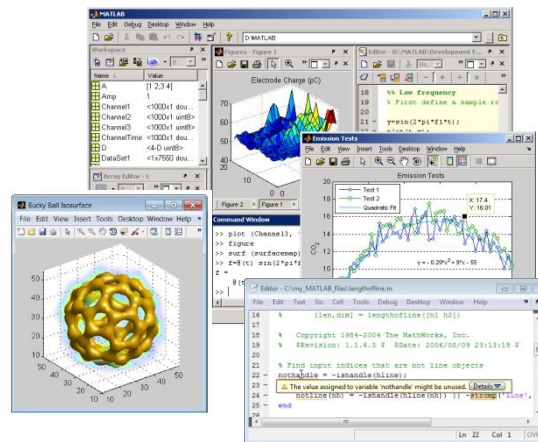
operator / on type int 32
left: 10
right: [-21474855 .. -1]
result: [-10 .. 0]

MathWorks概要

MATLAB

科学技術計算のための最先端の開発環境

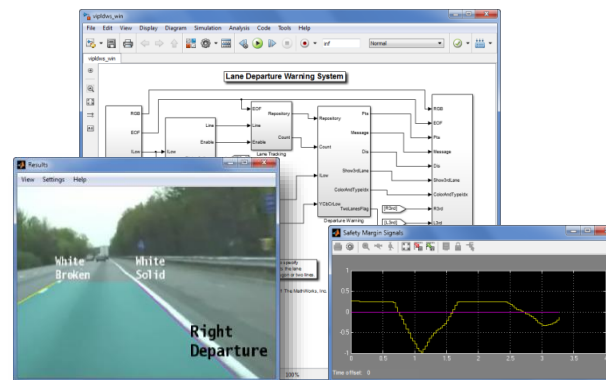
- 対話的なプログラミング環境
- アプリケーション固有の簡潔なプログラミング言語
- データの探索、解析、計算およびグラフィックス機能
- アルゴリズム開発、カスタマイズ可能な各種機能



Simulink

視覚的に理解可能なモデリング/シミュレーション環境

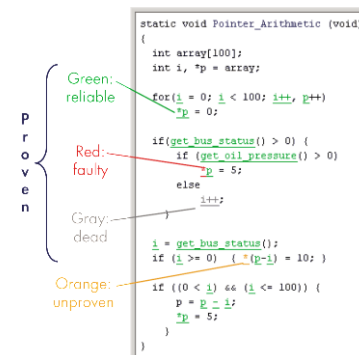
- ブロック線図シミュレーション環境
- 開発ツールの連携による統合された開発環境
- 自動コード生成による組込み実装



Polyspace

テストケースを必要としない静的なコード解析ツール

- ソースコード内にランタイムエラーが存在しないことを証明
- MISRA-Cなどのコーディング規約の適用
- IEC61508, ISO26262, DO-178など各種認証のレポート



- 本社:**
アメリカ、マサチューセッツ州
ネイティック市
- アメリカ:**
カリフォルニア、ミシガン
テキサス、ワシントン DC
- ヨーロッパ:**
フランス、ドイツ、イタリア
スペイン、オランダ
スウェーデン、スイス、イギリス
- アジア-パシフィック:**
オーストラリア、中国、インド、韓国
日本(2009年7月1日設立)
- 代理店 24カ国**

Polyspace 沿革

- 1999年に PolySpace が製品化され、2007年に MathWorks 製品に加わる
- 1996年から実用化の段階に
 - Ariane 5 のソフトウェアでの実証
- 300以上のユーザーにおいて2000以上のライセンスが使用される

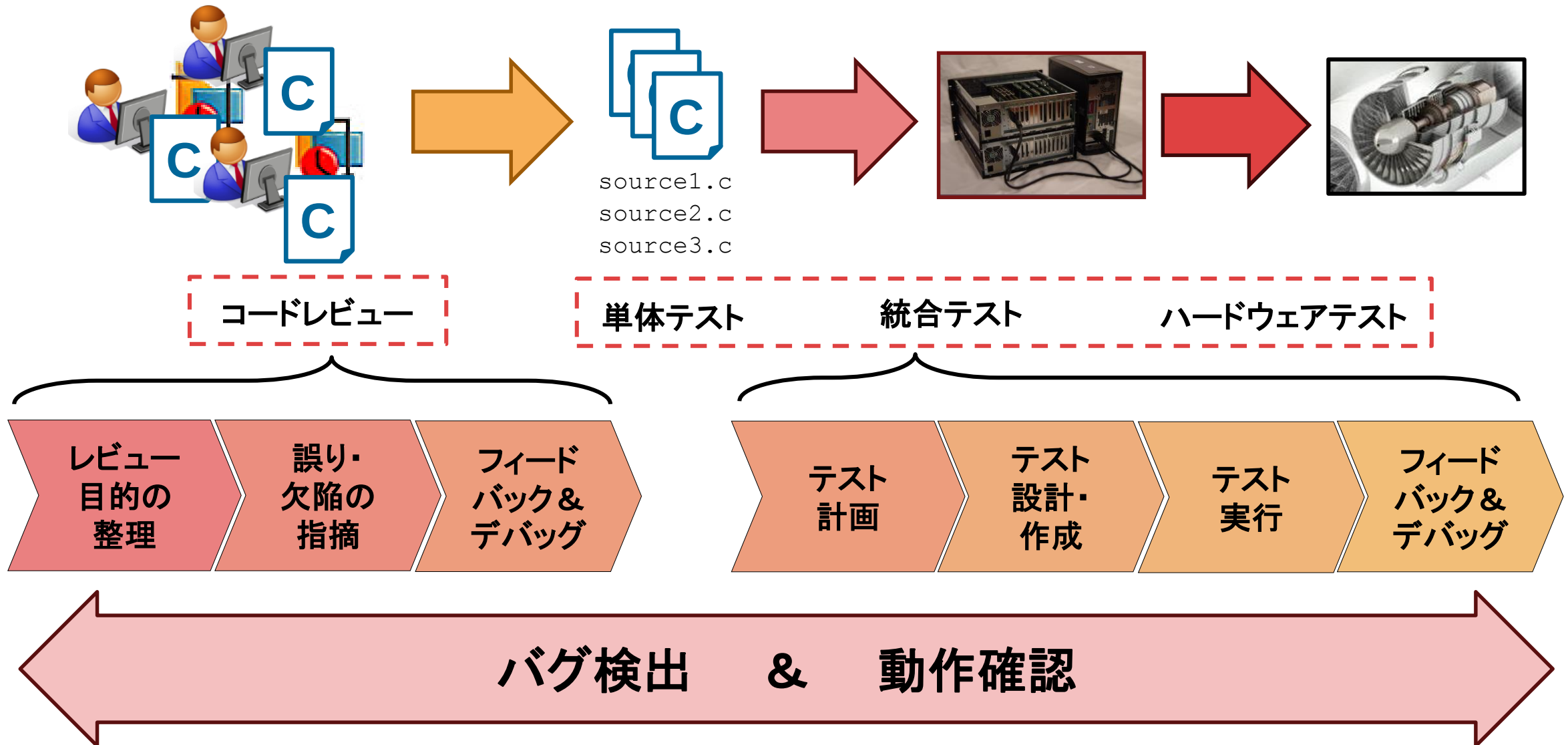


Polyspace ユーザーリスト

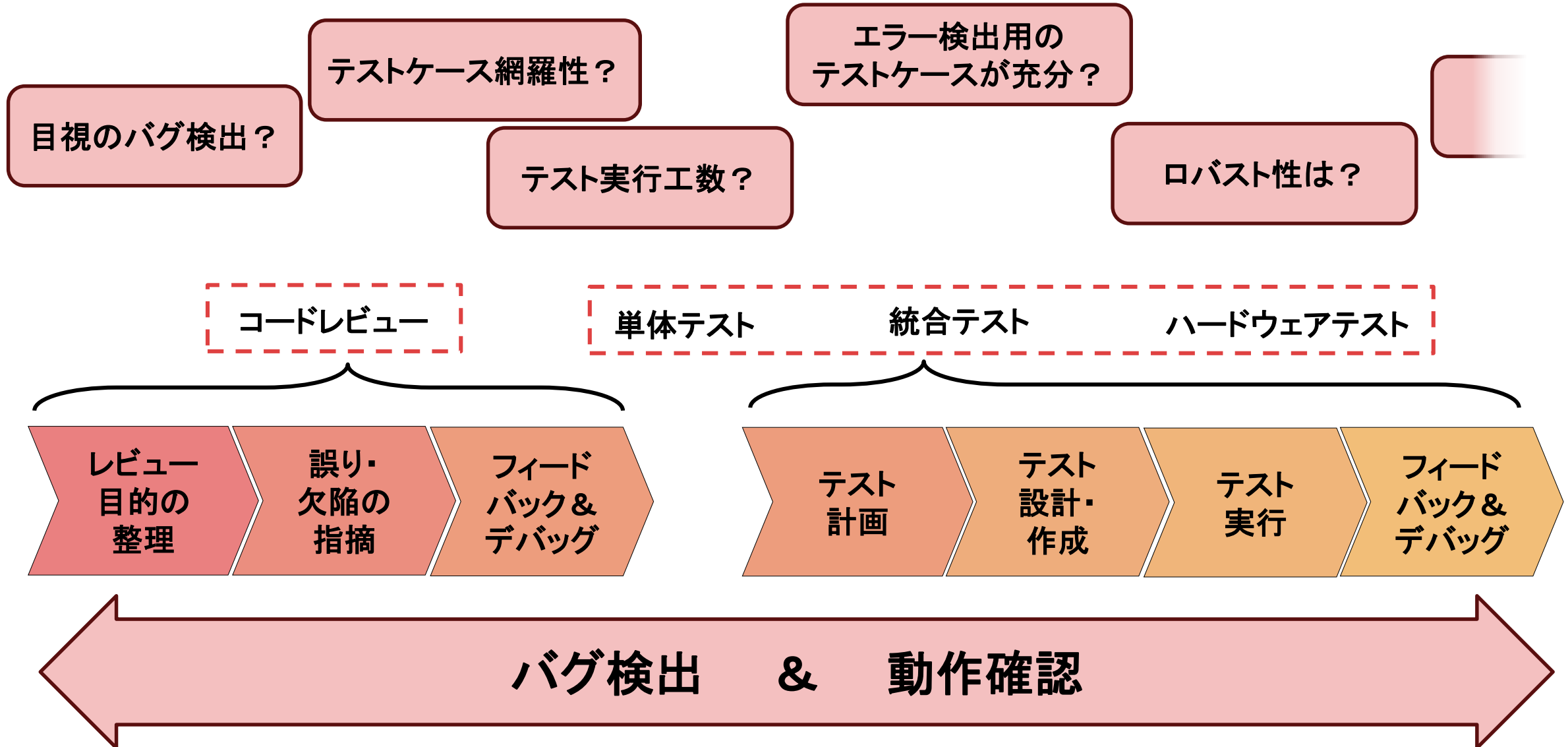
- 航空宇宙・防衛
- 自動車
- 産業装置
- 交通・運輸
- 家電製品
- 医療機器



一般的な検証プロセス

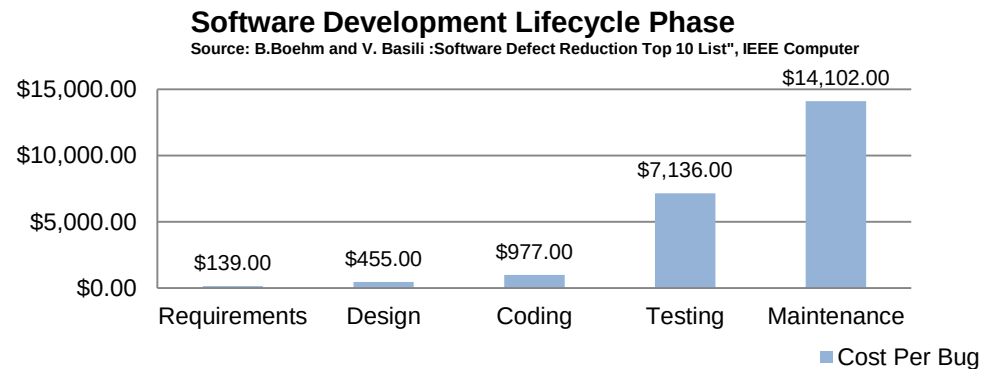
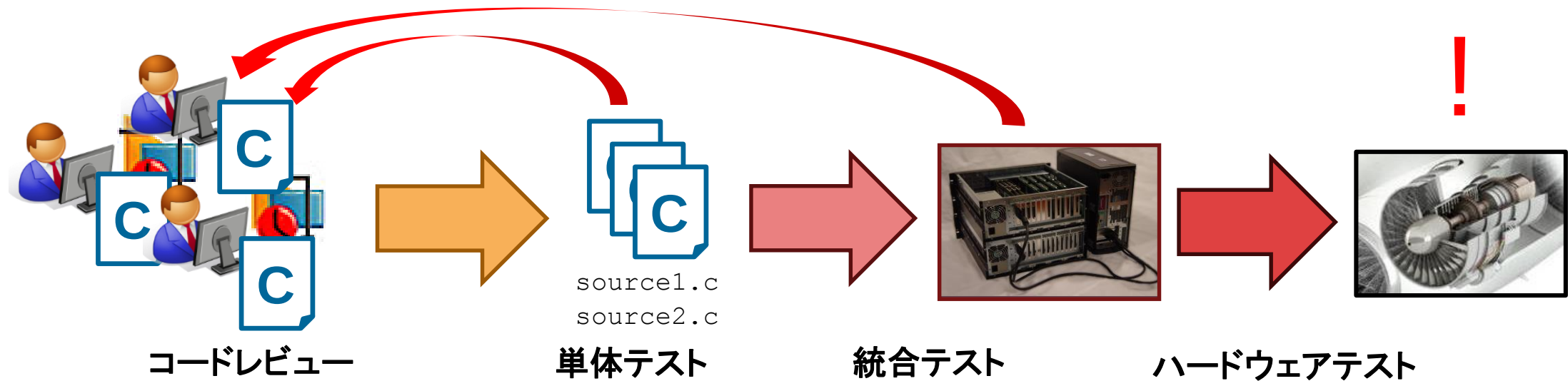


様々なチャレンジが存在



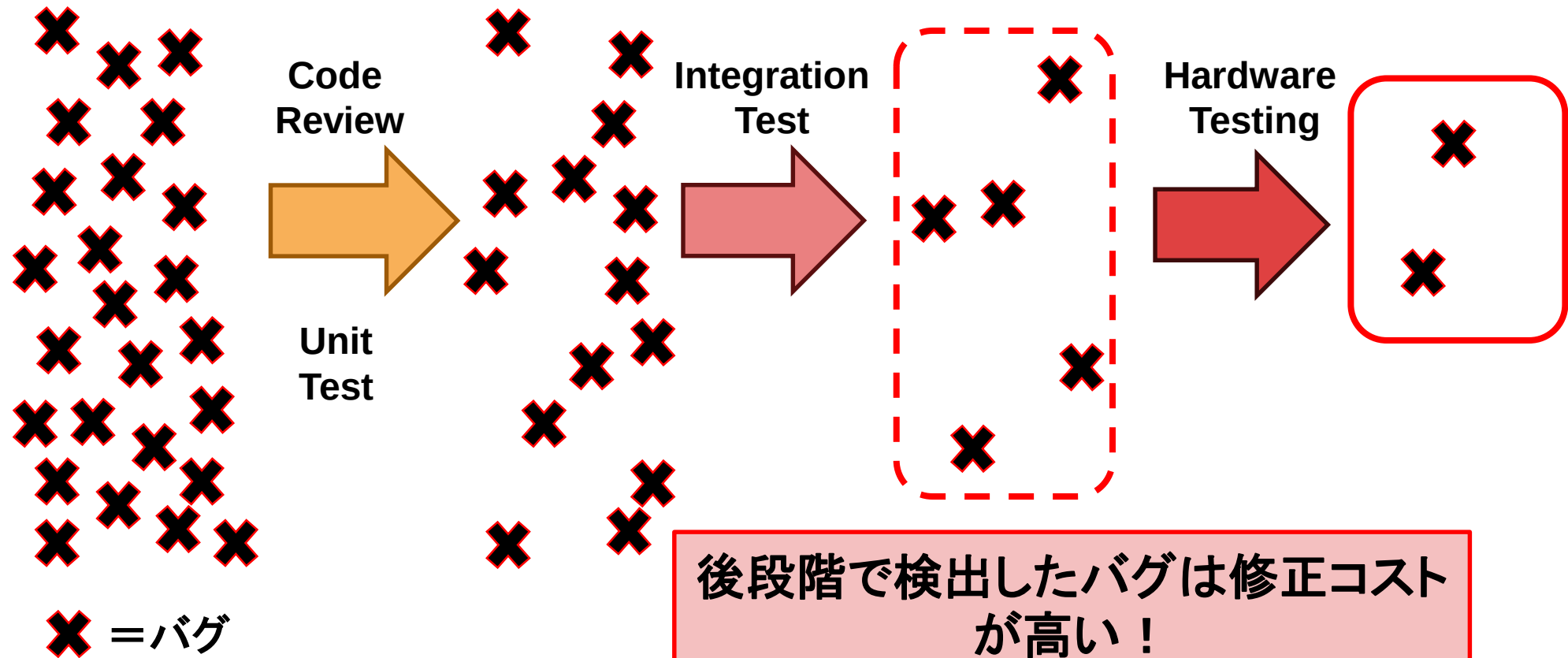
開発後期でのバグ検出の危険性

動的テスト完了後・製品リリース後にエラーが判明した！



例えばこのようなことが起こってはいないでしょうか？

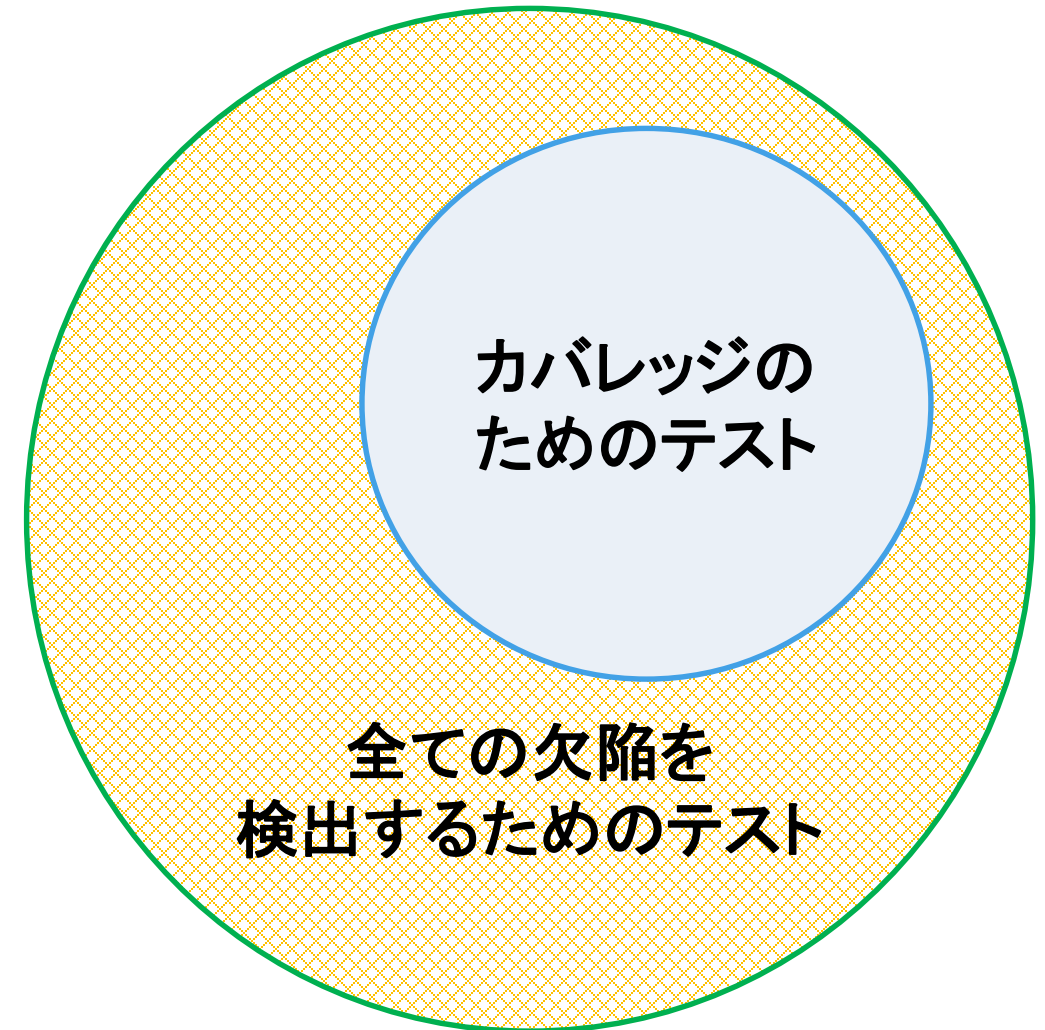
ソフトウェアバグが残されて検証を進めている可能性



後段階で検出したバグは修正コスト
が高い！
製品リリース後のバグはより危険！

テストのみでは足りません！

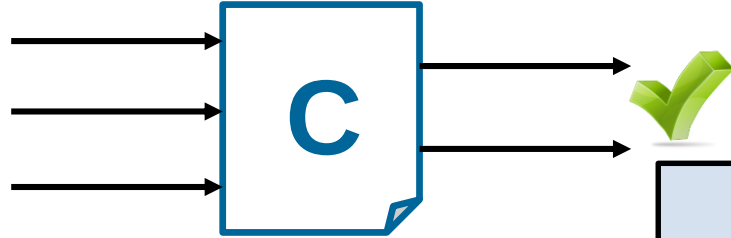
- テストの効果
 - 良いテスト設計により、機能的エラーを削除します
- 安全性
 - テスト後に未検知のランタイムエラーは障害を発生させます



形式手法に基づく静的解析で、Polyspaceはソフト安全性を確保

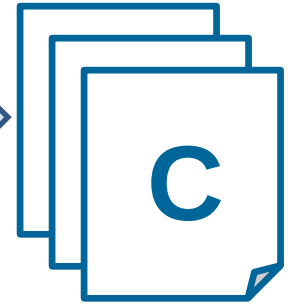
コード検証の目的

要求仕様通りの設計かを確認



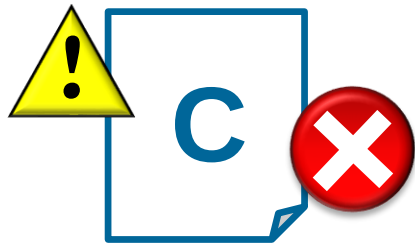
要求仕様に問題ないことを確認

PW01 スイッチ操作	
PW01 -1	運転席のダウンスイッチ押下により
PW01 -1-1	障害物を検知した場合は動作命令
PW01 -1-2	窓の下端到達を検知した場合は重
PW01 -1-3	ダウンスイッチはアップスイッチ、ニ
PW01 -1-4	ダウンスイッチが1秒以上継続して



動的検証

不具合(ゼロ割等)がないことを確認



静的検証

コーディングルール準拠の確認



適切な検証手法を使用してプロセスを効率化

Polyspaceの静的解析の種類

ファイル数
リカーション
循環的複雑度
関数コール
行数

コードメトリクス



コーディングルール
チェック

```
191 {  
192     int x = random_int();  
193     int y = random_int();  
194  
195     if (x > y) {  
196         x = x + 1;  
197     }  
198  
199     if (x < 0) {  
200         x = x + 1;  
201     }  
202     x = y;  
203 }
```

Polyspace Bug Finder

バグ検出

```
191 {  
192     int x = random_int();  
193     int y = random_int();  
194  
195     if (x > y) {  
196         x = x + 1;  
197     }  
198  
199     if (x < 0) {  
200         x = x + 1;  
201     }  
202     x = y;  
203 }
```

Polyspace Code Prover

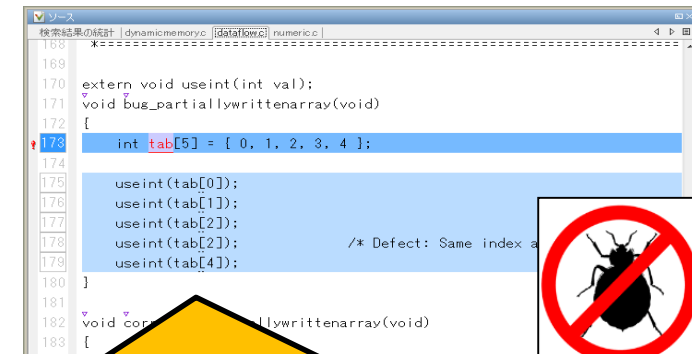
コード証明



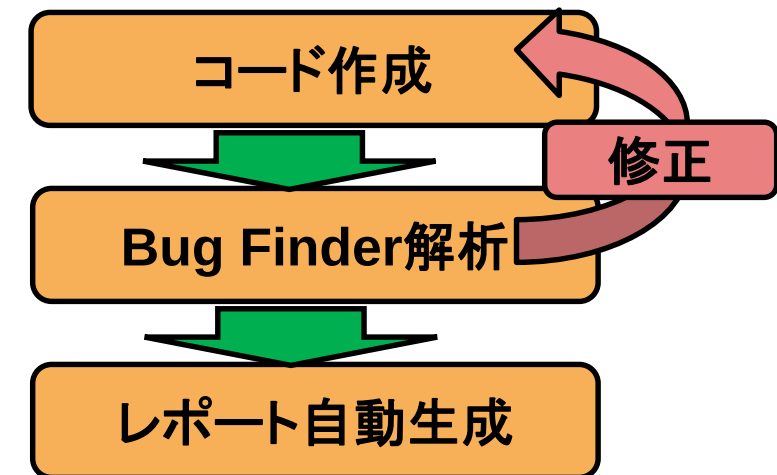
Polyspace Bug Finderのスピーディー解析

- **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！



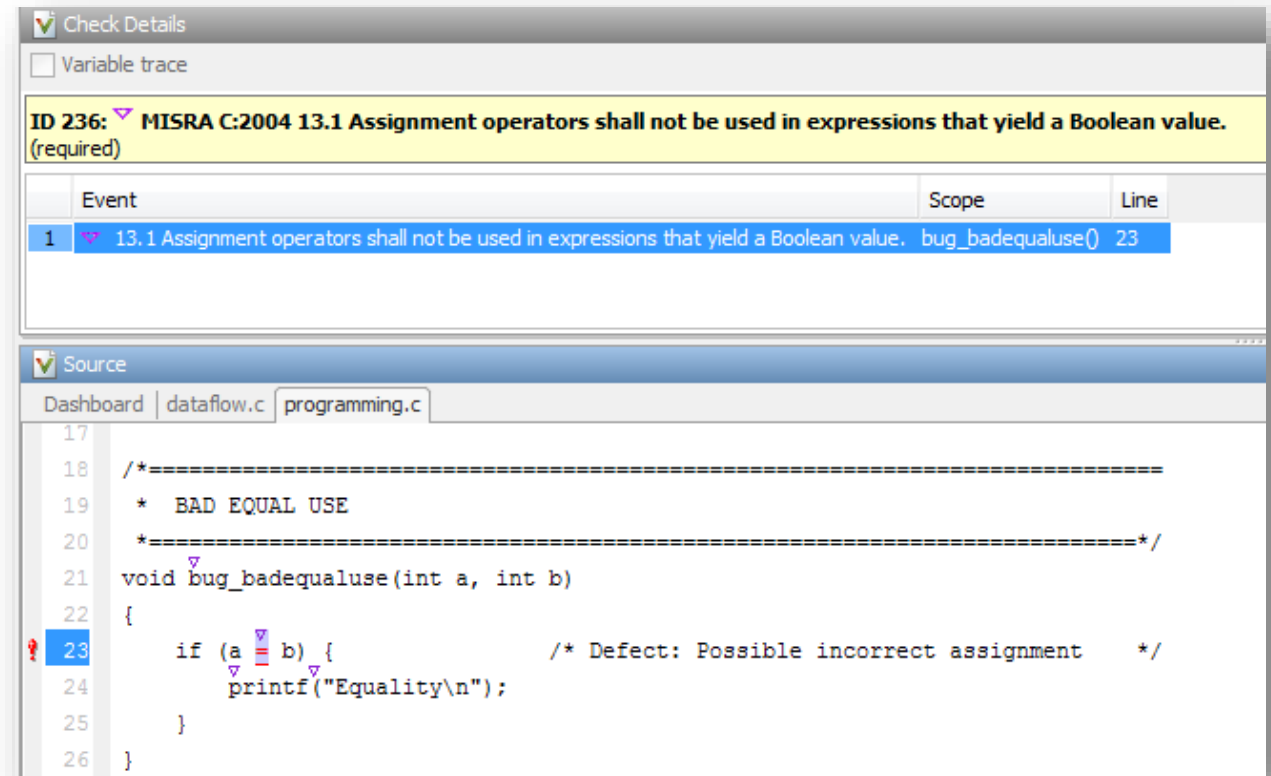
時間を掛けずに大部分のバグを識別



素早い欠陥検出・修正を可能とするPolyspace Bug Finder

コーディングルールの確認

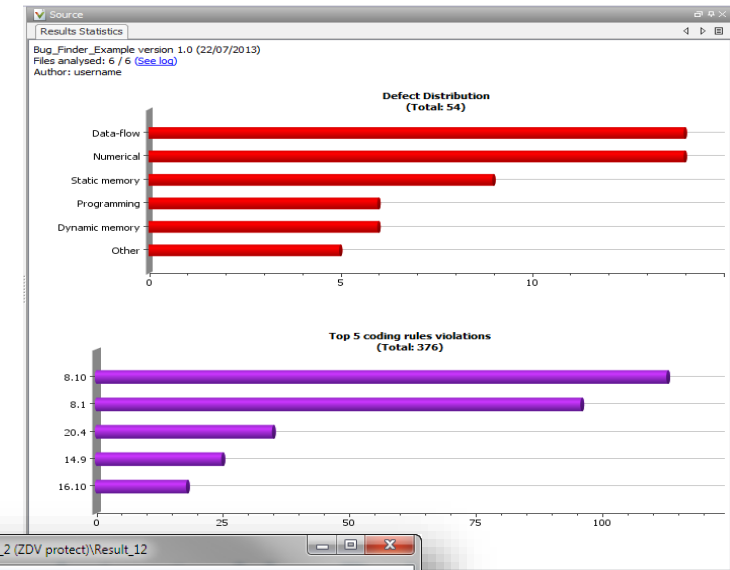
- MISRA C[®] チェッカー
- MISRA AC AGC
 - 自動生成コード用
- MISRA C++ チェッカー
- JSF++ チェッカー
- カスタマイズ
 - ルールのオン・オフ設定
- カスタムルール



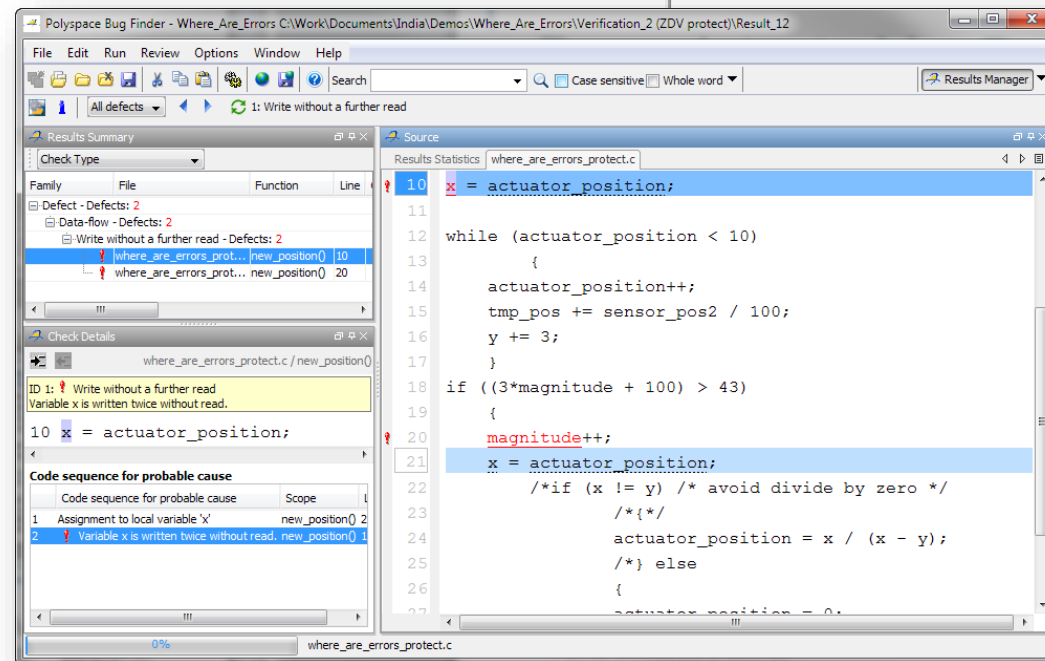
Polyspaceコードルールチェックによりエラーの予防可能

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 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

グリーン: 正常

ランタイムエラーが存在しない

レッド: エラー

実行される度にランタイムエラー

グレー: デッドコード

無実行

オレンジ: Unproven

条件によってランタイムエラー

パープル: Violation

MISRA-C/C++, JSF++

変数値範囲

ツールチップ

```
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;
    }
}
```

variable 'i' (int32): [0 .. 99]
assignment of 'i' (int32): [1 .. 100]

Polyspaceは全ての実行パスの結果を証明する！

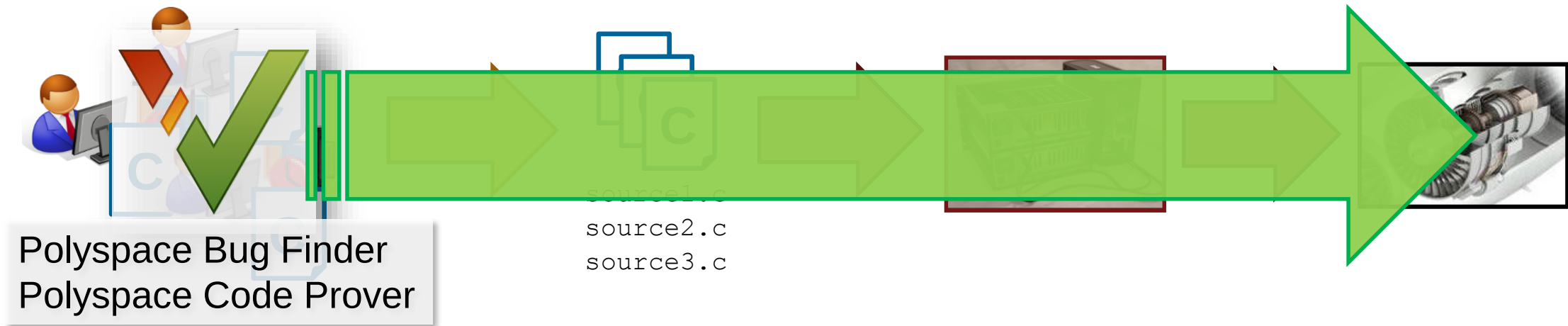
Polyspace Code ProverのKey Point

- “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

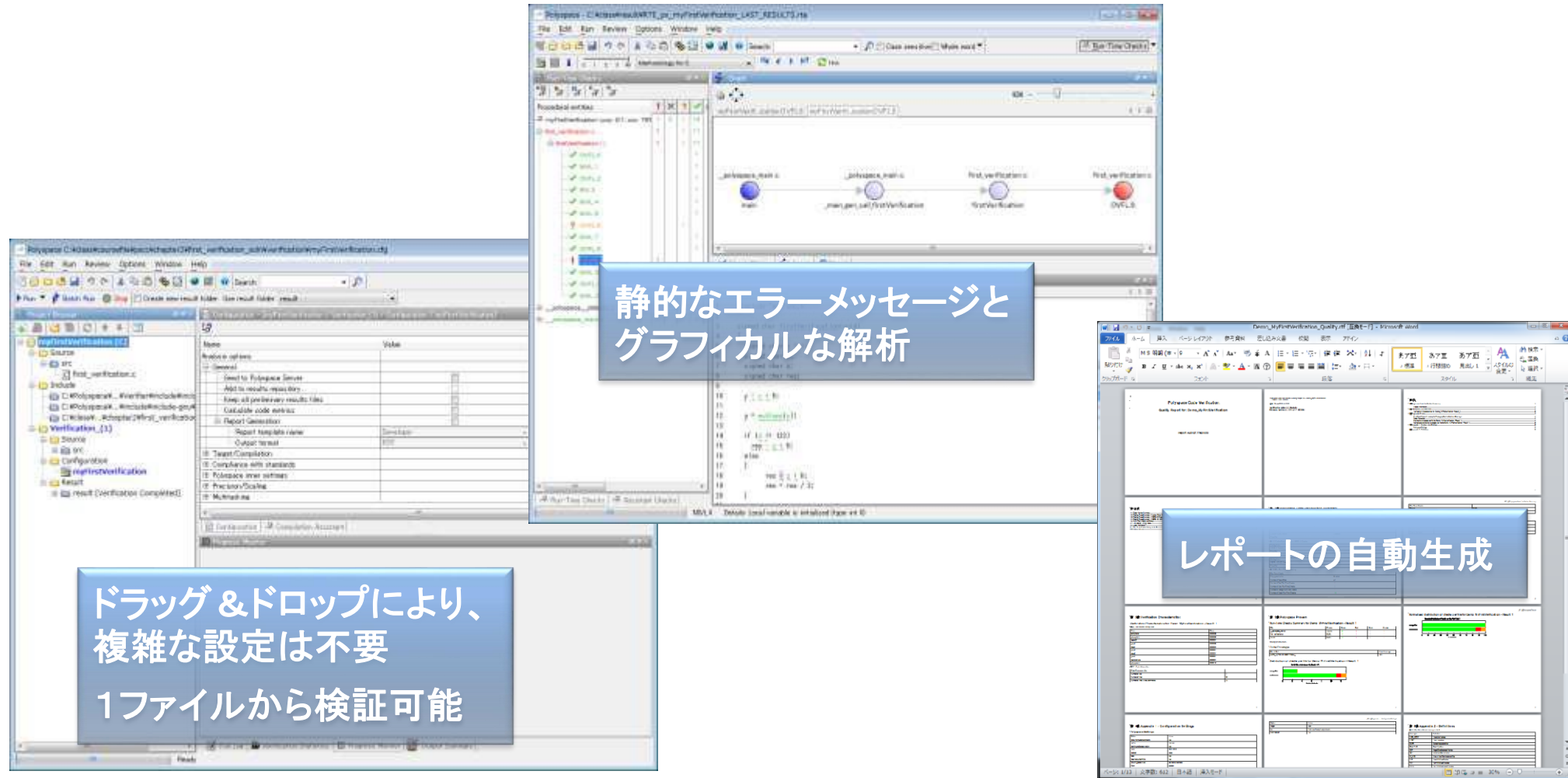
不具合の不在をPROVEする事がPolyspaceの特徴

Polyspaceのメリット

早期段階でのエラー検出・コード証明により
スムーズな流れを実現



スピーディーな静的検証の実現



静的なエラーメッセージと
グラフィカルな解析

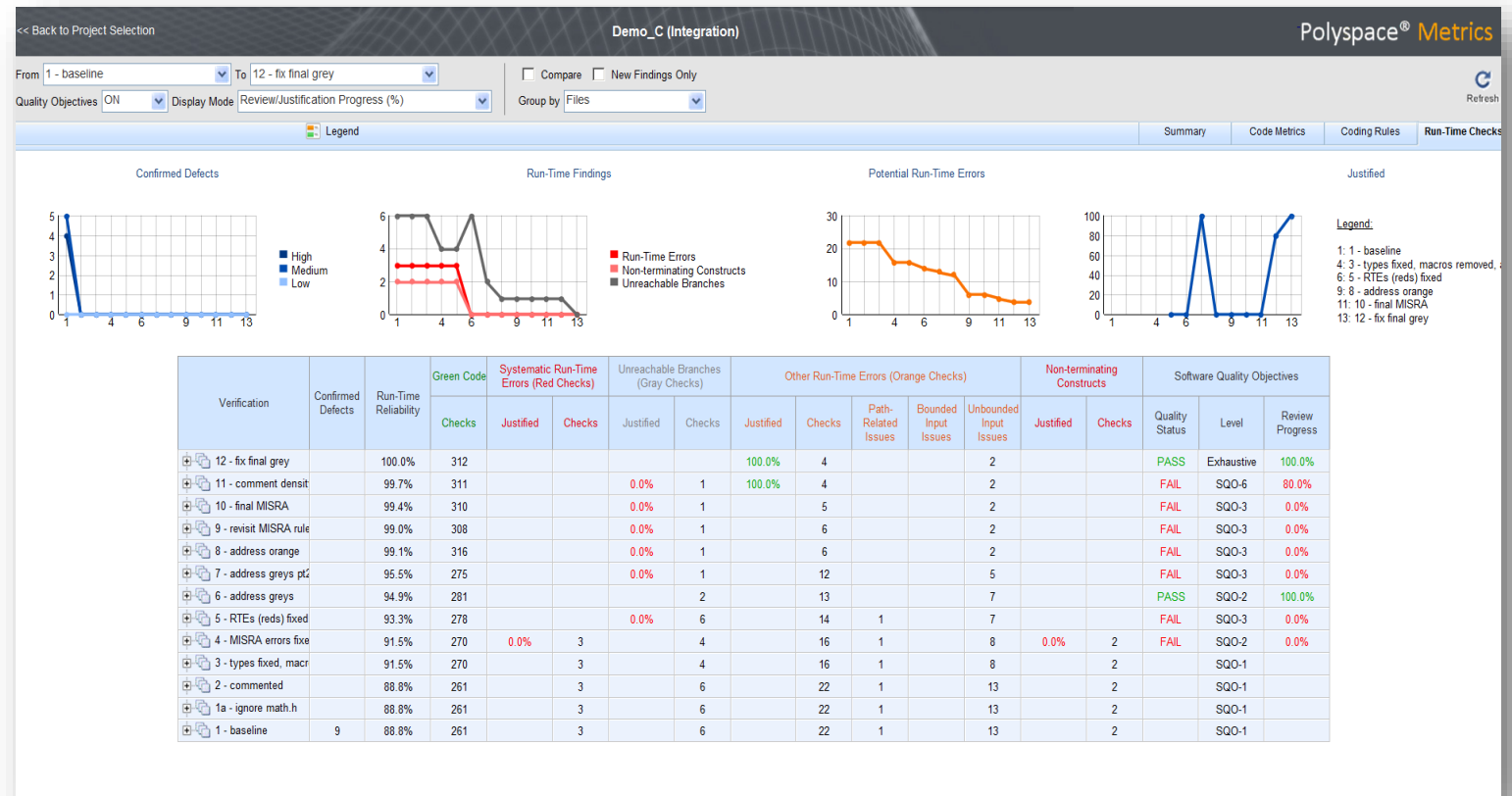
ドラッグ&ドロップにより、
複雑な設定は不要
1ファイルから検証可能

レポートの自動生成

重要な検証を確保しつつ、検証工数を大幅縮小

Polyspace Dashboard: 解析結果オーバービュー

- 品質トレンドの確認
- 目標に対するPass/Fail判定
- インクリメンタル結果表示



規格準拠に向けて

- 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
 - DO-178B
 - DO Qualification Kit
 - IEC 61508, ISO 26262, EN 50128
 - IEC Certification Kit



Polyspace製品で目標規格達成をアシストします！

Polyspaceユーザー事例



PCCN
Controls



Introducing Polyspace into the Software Development Process

Eileen Davidson
Ford Motor Company

12-May-2015

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 SENTRY-100, a noninvasive glucose monitoring system.



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。』

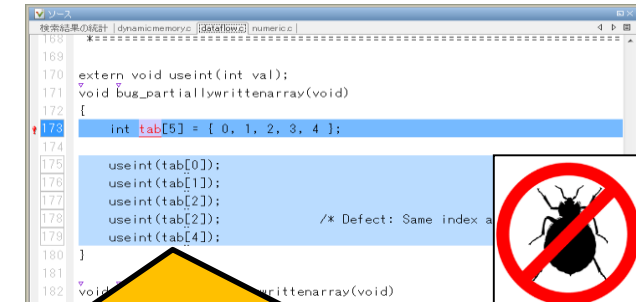
日産自動車 菊池様



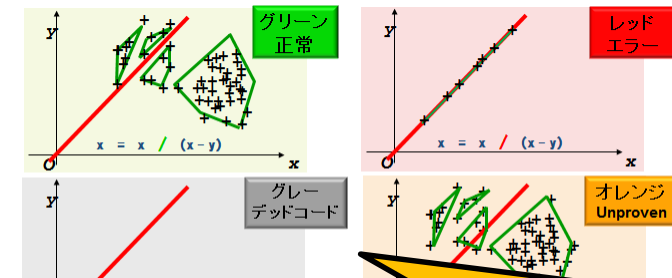
Polyspace ソースコード静的検証 ～まとめ～

- Polyspace Bug Finderで
素早くコードの欠陥を検出！
早期段階で不具合の修正が可能！
- Polyspace Code Proverで
クリティカルシステムにランタイム
エラーの有無を証明！
- 両ツールを使用して
効率的にソフトウェアの品質を
管理・向上！

Simulink環境から実行可能
ファイル・プロジェクト単位で使用可能

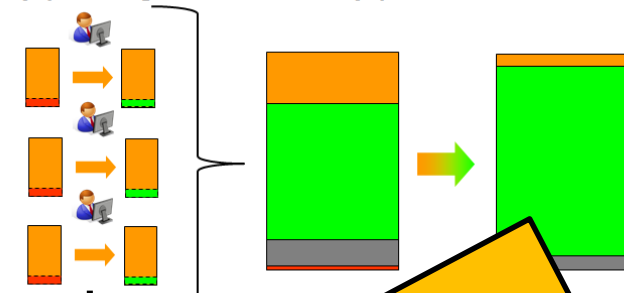


時間を掛けずに大部分のバグを識別



形式手法の解析によるコード信頼性

Polyspace Bug Finder → Polyspace Code Prover



安全なコードを確保して実機実験へ

トレーニング・コンサルティング サービス

■ トレーニング サービス

➤ 定期 トレーニング

- ✓ 東京、名古屋、大阪にて定期開催
- ✓ 基礎コース(11)、応用コース(11)、
専門コース(4)をご提供

➤ オンサイト トレーニング

- ✓ お客様サイトにて開催
- ✓ ご要望に応じて3つのレベルでカリキュラム
のカスタマイズが可能

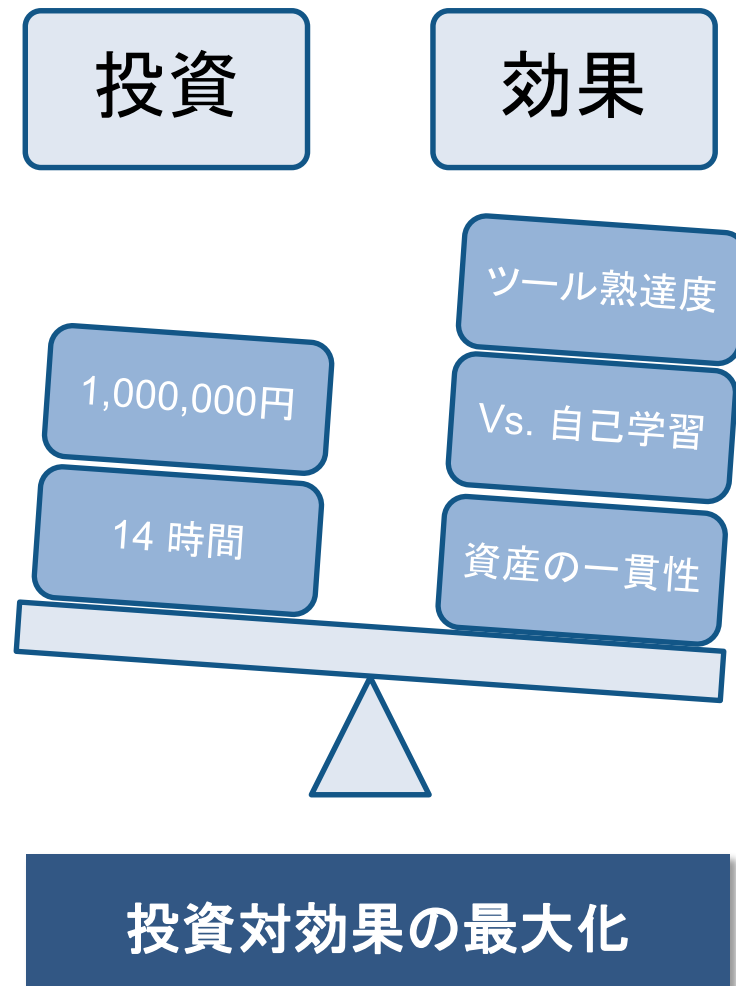
■ コンサルティング サービス

➤ カスタム “Jumpstart”

- ✓ 顧客モデルをベースにした短期集中型ツール導入サポート

➤ Advisory Service

- ✓ 顧客Project に合わせた中長期アドバイザリサービス



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

© 2016 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.

付録: 検出可能項目

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 use 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

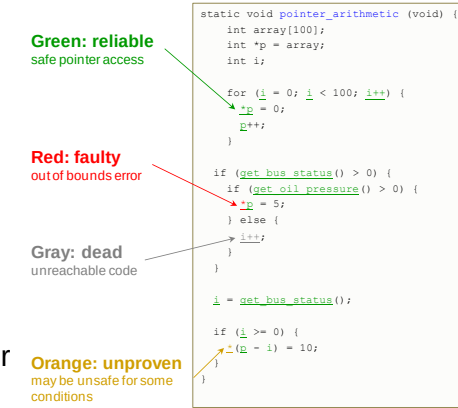
Polyspace Code Proverによるランタイムエラー有無の証明

C run-time checks

- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow (left shift on signed variables, float underflow versus values near zero)
- Initialized Return Value
- Shift Operations (shift amount in 0..31/0..63, left operand of left shift is negative)
- Illegal Dereferenced Pointer (illegal pointer access to variable of structure field, pointer within bounds)
- Correctness Condition (array conversion must not extend range, function pointer does not point to a valid function)
- Non-Initialized Pointer
- User Assertion
- Non-Termination of Call (non-termination of calls and loops, arithmetic expressions)
- Known Non-Termination of Call
- Non-Termination of Loop
- Standard Library Function Call
- Absolute Address
- Inspection Points

C++ run-time checks

- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow
- Shift Operations
- Pointer of function Not Null
- Function Returns a Value
- Illegal Dereferenced Pointer
- Correctness Condition
- Non-Initialized Pointer
- Exception Handling (calls to throws, destructor or delete throws, main/tasks/C_lib_func throws, exception raised is not specified in the throw list, throw during catch parameter construction, continue execution in __except)
- User Assertion
- Object Oriented Programming (invalid pointer to member, call of pure virtual function, incorrect type for this-pointer)
- Non-Termination of Call
- Non Termination of Loop
- Absolute Address
- Potential Call
- C++ Specific Checks (positive array size, incorrect typeid argument, incorrect dynamic_cast on reference)



付録: Polyspace Code Proverの抽象解釈によるコード証明

ランタイムエラーフリーを確保するために

$$x = x / (x - y)$$

- ランタイムエラーの可能性として
 - x や y は初期化されているか
 - 各演算でオーバーフローは発生するか
 - ゼロ除算は発生するか



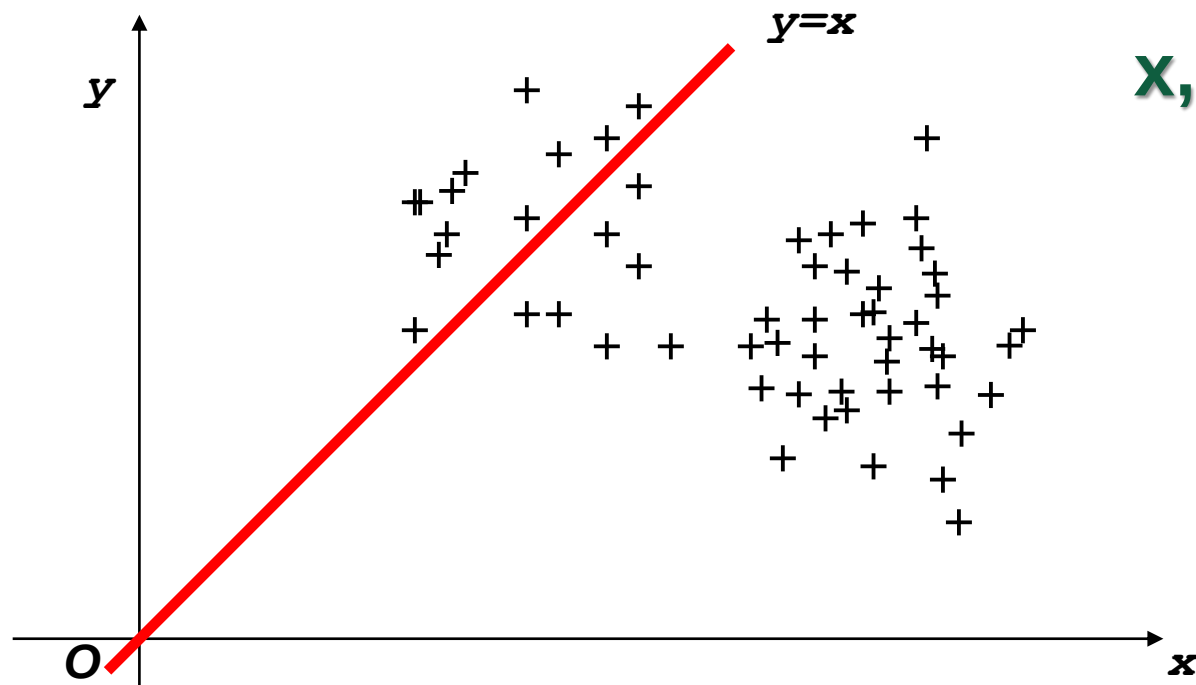
簡単なコードでも検証は複雑

32bit int 型の2変数の値の組み合わせは 2^{64} 通り

抽象解釈コード検証

ゼロ割の検証

$$x = x / (x - y)$$



x, y の取り得る値

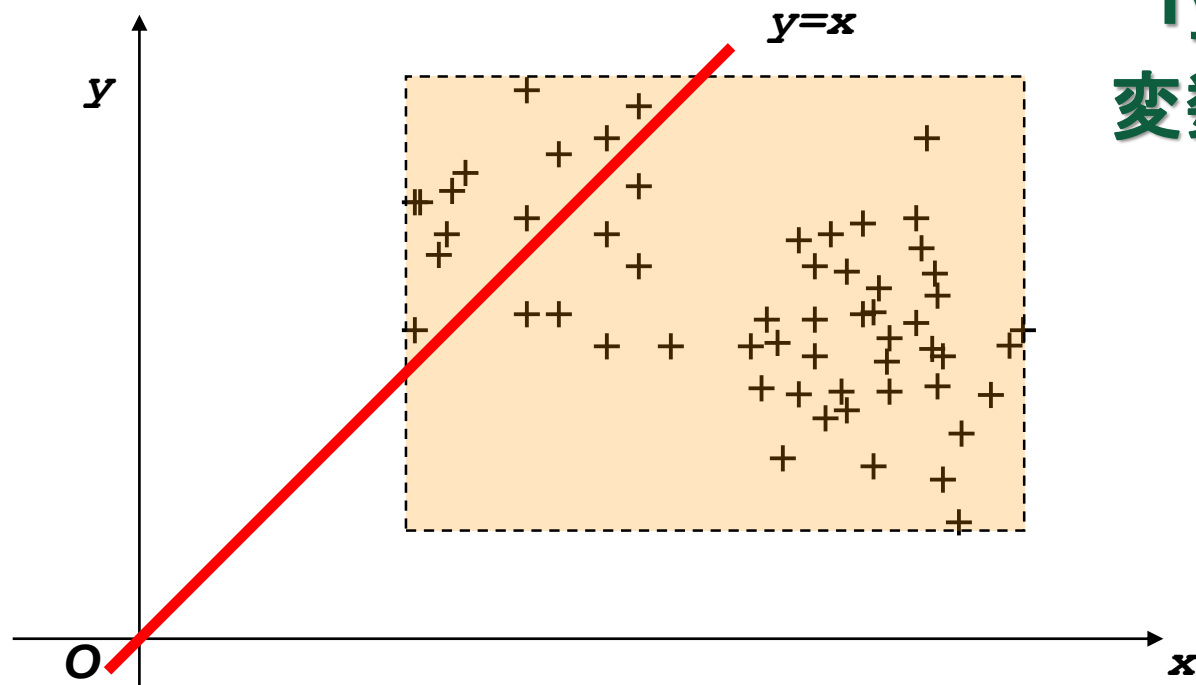
理想: 全てのテストケースを実行

現実: 時間的に無理

抽象解釈コード検証

ゼロ割の検証

$$x = x / (x - y)$$



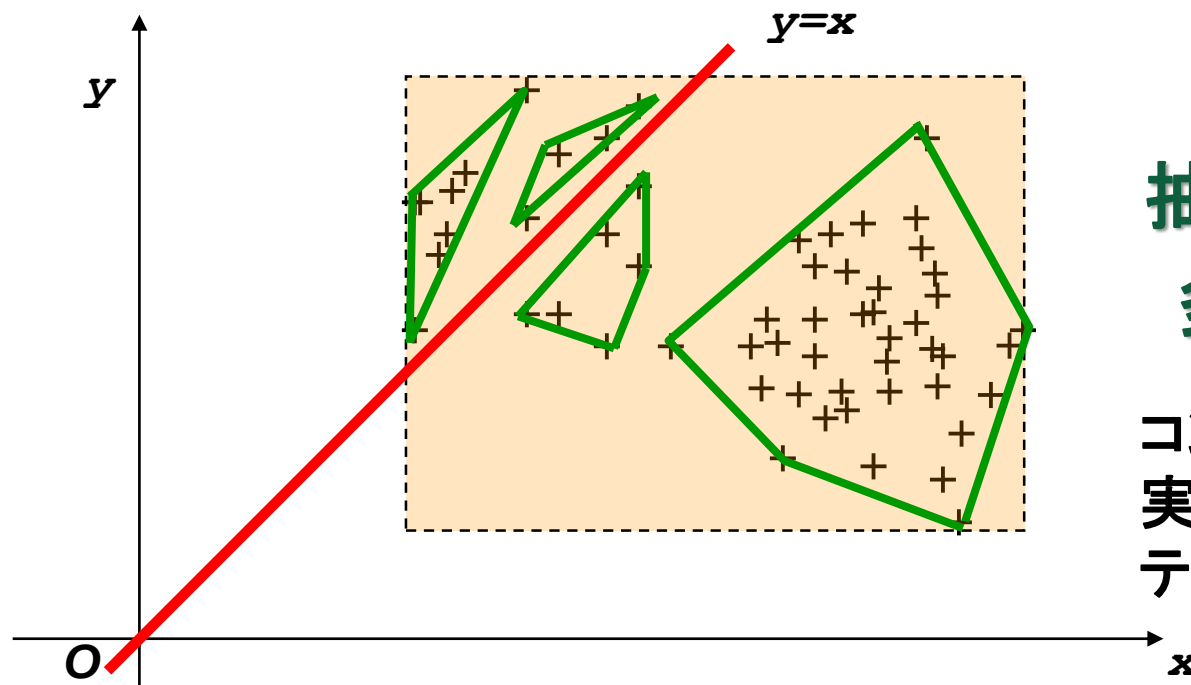
Type Analysis
変数レンジの近似

Type Analysisの正確度は不十分

抽象解釈コード検証

ゼロ割の検証

$$x = x / (x - y)$$



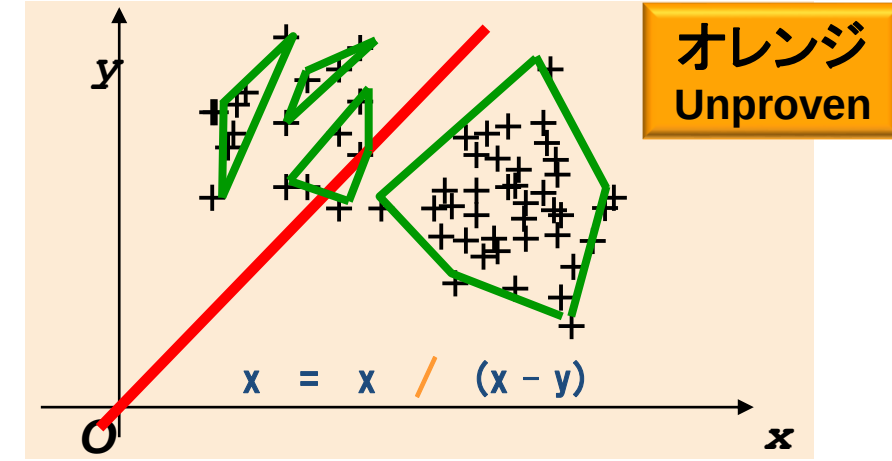
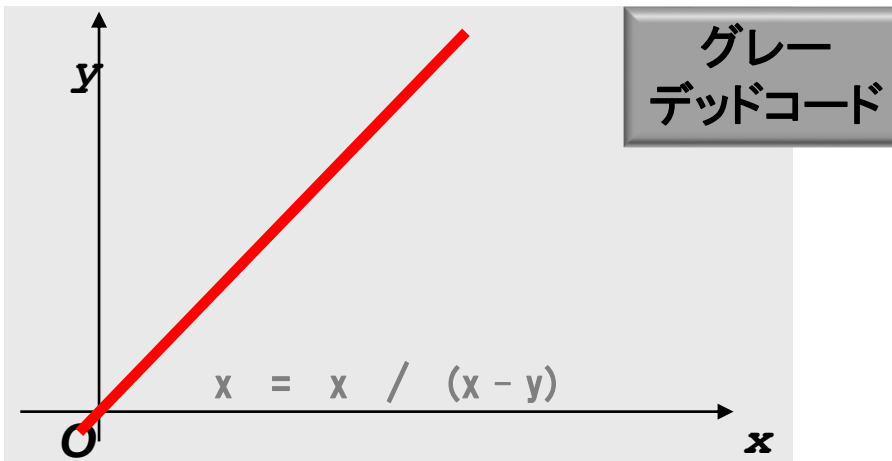
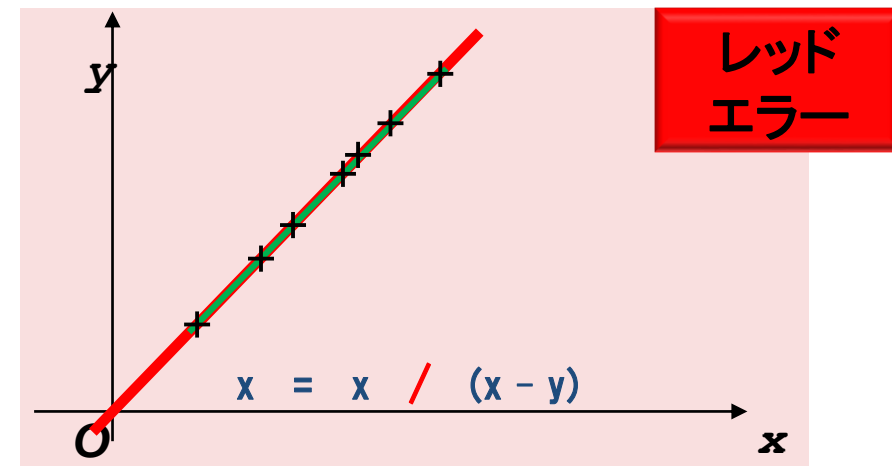
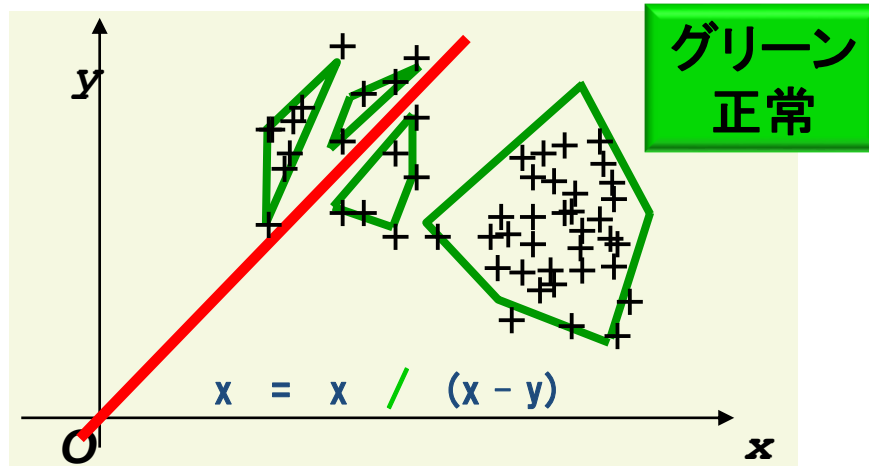
**Abstract
Interpretation**
抽象解釈による
多角形で近似

コンパイル不要
実行不要
テストケース作成不要

Abstract Interpretationで正確に全てのケースを網羅

検査結果の分類

$$x = x / (x - y)$$

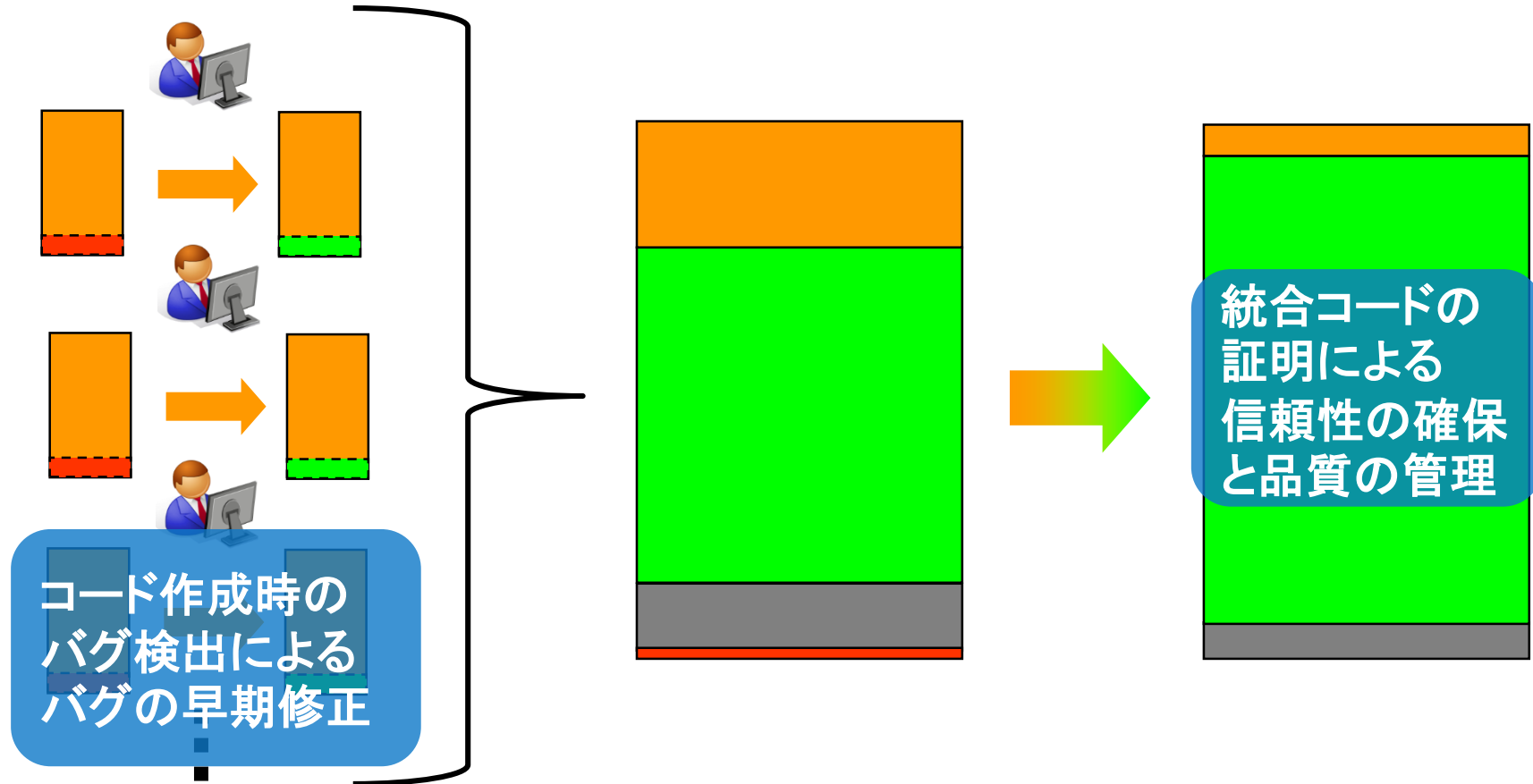


Polyspaceによる検査結果の分類

付録:その他

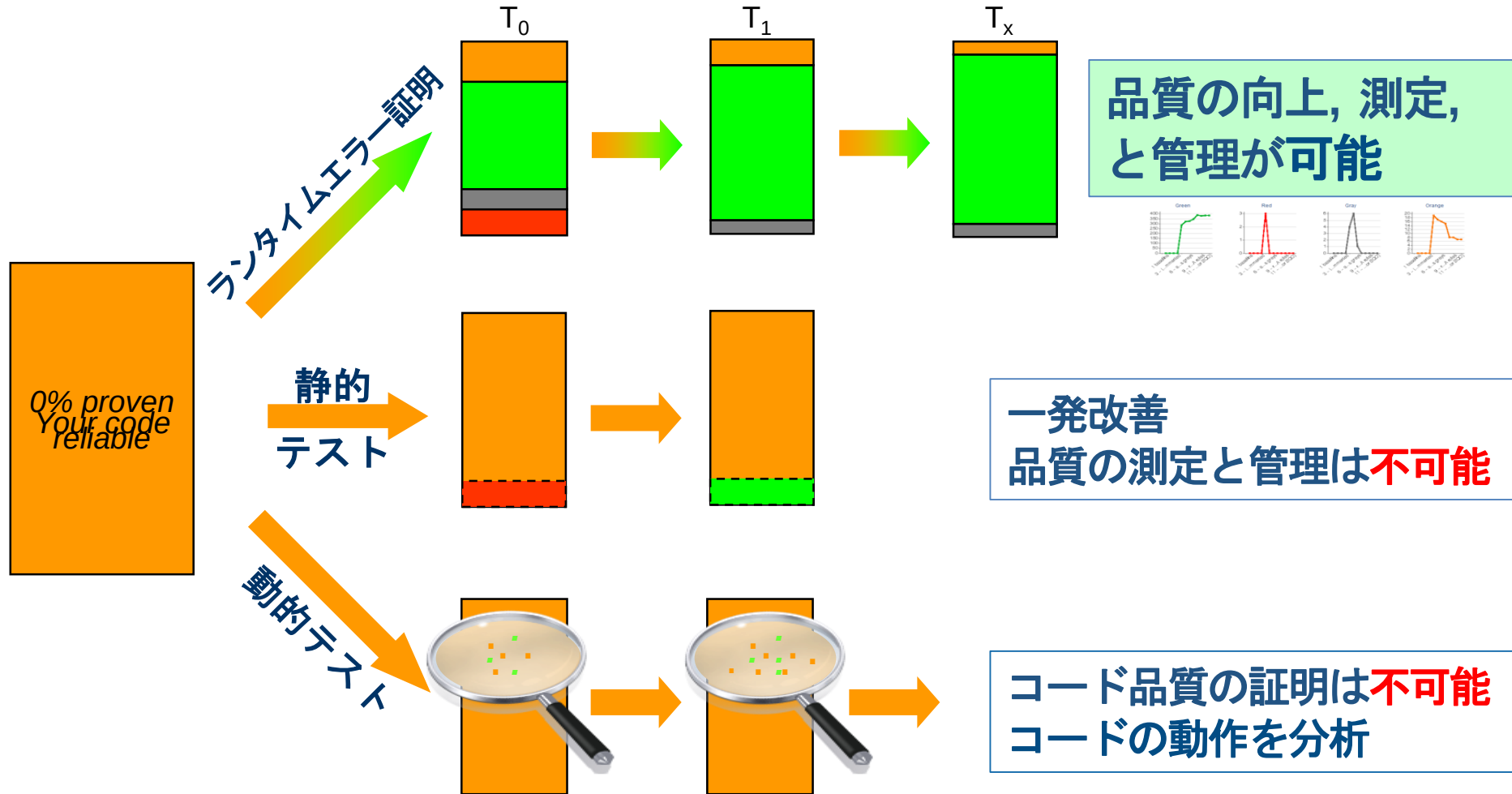
信頼性の確保に向けて

Polyspace Bug Finder → Polyspace Code Prover



Polyspace Bug FinderとCode Proverを使い分けて効率的にコードの信頼性を測定・確保

なぜPolyspaceが良いか？



Polyspaceでコードがエラーフリーの信頼性を確保