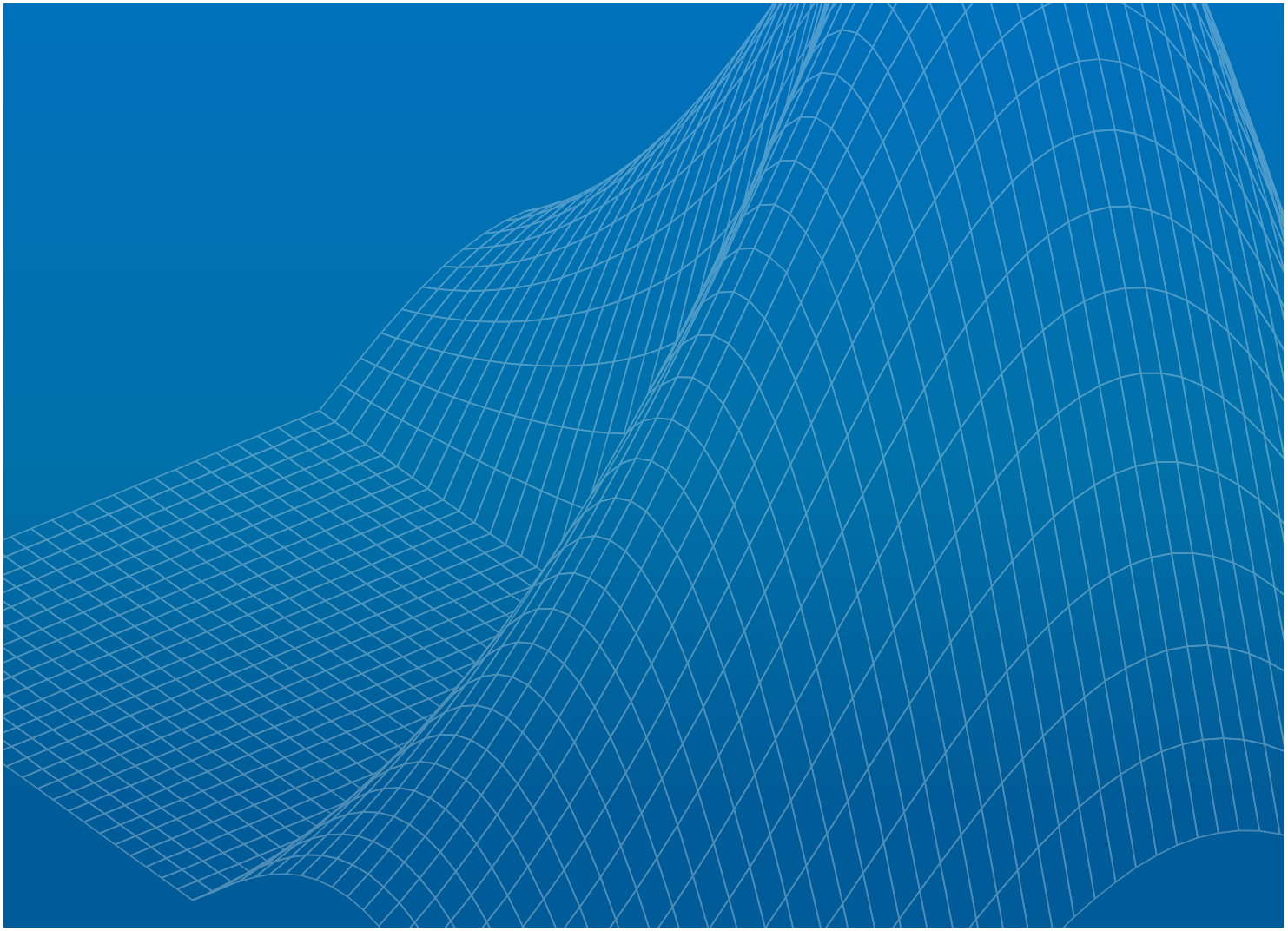

Polyspace製品を使用した包括的な静的解析

組み込みソフトウェアの検証における課題を解決するソリューション

WHITE PAPER

MathWorks®

はじめに

近年、組み込みソフトウェアの検証は非常に困難になっています。開発スケジュールのプレッシャーや複雑度の増加により、通常の手動検証手法ではソフトウェアのバグやランタイムエラーを漏れなく把握することは難しくなっております。コードレビューは工数が多くかかり、規模・複雑度が増加したアプリケーションに対する適用は非現実的です。動的テストによるエラー検出はエンジニアがテストケースを作成し、実行する必要があります。実行中にエラーが発生したら、エラーの原因を解析する手間が掛かります。そもそも、用意したテストケースで全てのエラーが見つけることはエラーを起こすテストケースを全て用意することであり、無理な提案です。

Polyspace® 静的解析ツールは検証プロセスを改善します。組み込みソフトウェアのバグ検出と、形式手法である抽象解釈に基づき、ソフトウェアの安全性を確保することが可能です。Polyspace Bug Finder™ は組み込みC/C++ソフトウェア内の実行時エラー、データフローの問題と、その他の欠陥を検出します。静的解析により、ソフトウェアの制御フロー、データフロー、関数呼び出し間の振る舞いを解析します。Polyspace Bug Finderは開発プロセス上流におけるバグ検出と修正作業を支援します。

Polyspace Code Prover™ はより高精度の解析を提供します。C/C++ソースコード内のオーバーフロー、ゼロ除算、配列の範囲外アクセスや、その他のランタイムエラーの存在・不在を証明することが可能です。抽象解釈により、数学的にコードの正確性を証明します。Polyspace Code Proverの解析結果は各操作を色付けしてランタイムエラーの存在・可能性・不在とデッドコードを明確に表示します。

抽象解釈により、Polyspace Code Proverはソフトウェアの動的な特性を検証し、ランタイムエラーが発生しないソフトウェアであることを確認できます。高精度にコードのロバスト性の証明とエラーの早期検出を可能とします。組み込みソフトウェアの動的特性を検証することで、抽象解釈は全てのソフトウェアの振る舞いや可能な入力データを網羅します。コードの正確性を証明し、コードの信頼性を確保します。バグ検出及びコード証明ツールの活用により、開発費用の低減と同時に高信頼性の組み込みソフトウェアの納入を迅速に行うことが可能です。

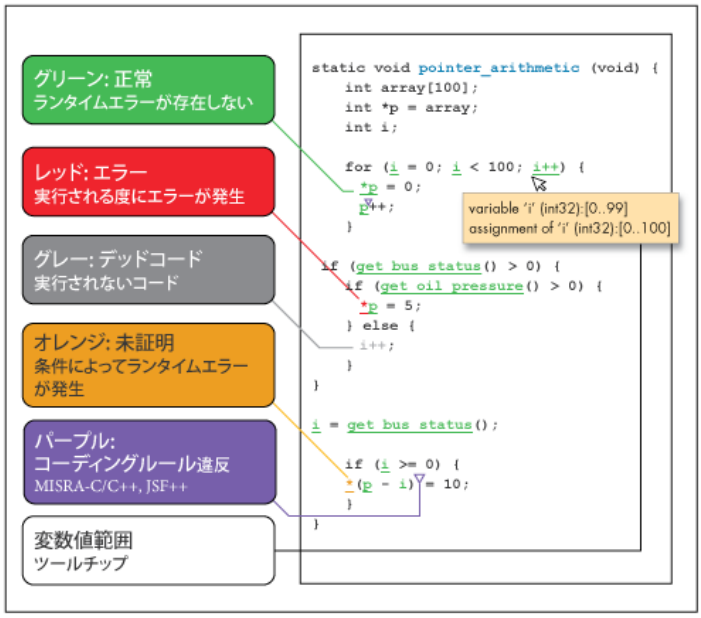

図1: Polyspace Code Prover検証によるランタイムエラー証明

組み込みソフトウェアテストに対するチャレンジ

近年、ソフトウェアの重要度はますます高まっています。ターゲットプロセッサの低価格化により、ソフトウェアに依存する部分が増加しています。その上、より高機能・高性能なシステムが求められ、ソフトウェアがこのような要望に対応する必要があります。このソフトウェアはシステムの中心となり、非常に厳しく評価され、問題がないような品質が要求されます。なぜなら、ソフトウェアの不具合により、製品のリコール、ブランド名・信頼の劣化が発生する可能性があります。クリティカルシステムの場合、物的障害や人命に関わる可能性もあります。

ソフトウェアの複雑化に関わらず、納入スケジュール・プレッシャーは変わりません。要求の変更、仕様書の更新、設計の見直しもあります。開発部隊はビジネス目標を達成し、工数低減する上に高品質の組み込みソフトウェアを開発する必要があります。このチャレンジはエンジニアリソースの不足により厳しくなります。

一つの解決案として、コード作成、設計、計測ツールを使用して作業を効率化します。このようなツールは開発の加速化を支援します。しかし、テスト・デバッグツールは組み込みソフトウェアの規模と複雑性のトレンドに追いついていません。そのため、組み込みソフトウェアのテスト作業は全体の開発コストの半分以上になる場合もあります。

早期のコード検証は時間とコストに対するチャレンジを解決する手法となります。開発の遅い段階で不具合が検出されるとデバッグの作業が複雑になります。何十万行、何百万行のコードをトレースして不具合の原因を解析する必要があり、多くの場合、コード作成者と異なるテストエンジニアが解析を行います。テストフェーズの後工程で不具合を検出した場合、コード作成時に不具合を検出した際と比較するとコストが10〜20倍の違いがあります [1]。

早期のコード検証のメリットは明らかになっていますが、現在はこのような検証は例外的に行われています。多くは、プロジェクトの最もプレッシャーが高いデッドライン近くに検証作業が行われます。この場合、最も検出、診断、修正が困難なランタイムエラーはそもそも検出されないこともあります。

このような数学的なコードエラーは潜在的エラーであり、通常の実行状況では発見されません。ゼロ除算、範囲外の配列アクセス、不正なポインター参照や、未初期化変数はこのようなエラーの一種です。ソフトウェアメンテナンスで発見される不具合の20〜50%はランタイムエラーです [2]。

想定外の動作、不正確な処理、又はプロセスの停止を発生させます。ランタイムエラーはソフトウェア信頼性に対する予想外の劣化の原因になる可能性があります。

一般のランタイムエラー検出手法の限界

通常のランタイムエラー検出・デバッグで使用されるツール・手法は古い技術をベースにしています。大きく分けて「手動によるコードレビュー」と「動的テスト」の二つのカテゴリーに分けることができます。この技法は全てのランタイムエラーを検出することはできません。

手動によるコードレビューは小規模なアプリケーションでは有効な場合もあります。大規模システムになりますと、必要な工数が現実的ではありません。経験豊富なエンジニアがソースコードをレビューして危険性を含むコード部分を指摘する作業は複雑・非包括的であり、高コストであるにも関わらず同じ作業を繰り返すことも困難です。ランタイムエラーがコードに存在しないことを明確にすることはコードレビューでは対応しきれません。

---

[1] Basilli, V. and B. Boehm. "Software Defect Reduction Top 10 List." Computer 34 (1), January 2001.
[2] Sullivan, M. and R. Chillarge. "Software Defects and Their Impact on System Availability." Proceedings of the 21th International Symposium on Fault-Tolerant Computing (FTCS- 21), Montreal, 1991, 2-9. IEEE Press.

動的テストはテストケースを作成する必要があります。通常の組み込みアプリケーションでは何千テストケースが必要な場合もあります。テストケース実行後、結果をレビューしてエラーの原因を分析する必要があります。この技法は40年近く大きく向上されていません。コード計測・コードカバレッジを用いて不具合の検出率を向上することは可能ですが、実行時に利用可能な値の範囲を全て網羅することは非現実的です。コードレビューと似て、多くの工数が必要です。テストケースが作成され実行可能な状況になるまで動的テストを行うことはできず、不具合の修正コストが一番高い開発プロセスの最後にテストを実行することが多くなります。

組み込みアプリケーションにおける動的検証による ランタイムエラー検出に関するチャレンジ

アプリ規模に並ぶ指数的に増加する検証コスト

コード量により平均的なエラー数があります。この数は控えめに1000行に2〜10エラーです。エラー検出の確率は行数が増加することで劣化します。コード規模が倍になることで、テスト工数を約4倍にしなければ同一の信頼性を得ることができません。

動的検証はエラーの症状を示すが、原因は示さない

検出される各エラーに対してエラーを再現して原因を分析する工数が必要です。開発の後期段階における不具合の原因解明はとても困難で時間が掛かる作業です。テスト作業の一部は自動化できますが、デバッグ作業は自動化できません。100行に2〜10エラーがあると考えた場合、5万行のアプリケーションは最小100エラーがあります。開発者が各エラーのデバッグで10時間が掛かる場合、アプリケーションのデバッグに1000時間が必要になります。

動的検証を行う際のコード計測

コード計測を行うことでプログラム実行中に不具合の観察が容易になります。しかしコードを計測するための工数もあり、オーバーヘッドを増加し、メモリー違反は共有変数の競合状態のようなエラーをマスクしてしまうこともあります。コード計測に基づく手法は実行したテストケースが不具合を発生する際のみに有効になります。

ランタイムエラー検出の有効性はテストケースに依存します

実行するテストケースは通常全ての可能なケースの一部のみ網羅します。そのため、多くのケースは確認されておらず、ランタイムエラーが未検出のままの可能性があります。

Polyspace Bug Finderによる開発の早期段階でのバグ検出

コードを作成する最中にバグ検査を行うことが推奨されています。ソフトウェアエンジニアは素早く、効率的なバグ検出技法を必要とします。Polyspace Bug Finderはコードモジュール、或いは全体のアプリケーションを解析する静的解析ツールです。Polyspace Bug Finderは形式手法を含む素早い静的コード解析技術を使用して、誤検出が少なく数式・データフロー・プログラミング・静的メモリー・動的メモリー・その他の欠陥を検出します。

Polyspace Bug Finderにより、欠陥はソースコード上に指摘され、欠陥の原因を分析するためのトレース情報もあります。MISRA-C/C++等のコーディングルールに対する違反も検出し、ルール違反を説明する詳細も確認できます。このような情報を使用して開発の早期段階でのデバッグ・修正作業を支援します。Polyspace Bug Finderはコマンドライン、ユーザーインターフェイス、Eclipse™ IDEから

の実行に対応しています。ビルド環境への統合による自動化も可能です。コード生成環境であるSimulink®、TargetLink®、IBM® Rational® Rhapsody®との統合にも対応しています。

Polyspace Bug Finderは検出される各欠陥に対して、原因の詳細を表示します。例えば、整数オーバーフローが発生する場合、オーバーフローに関係する行をトレースします。ソフトウェアエンジニアはこの情報を使用して適切な修正方法を分析できます。品質保障エンジニアは欠陥の詳細情報を使用して欠陥の分類を決め、次のアクションを指摘できます。

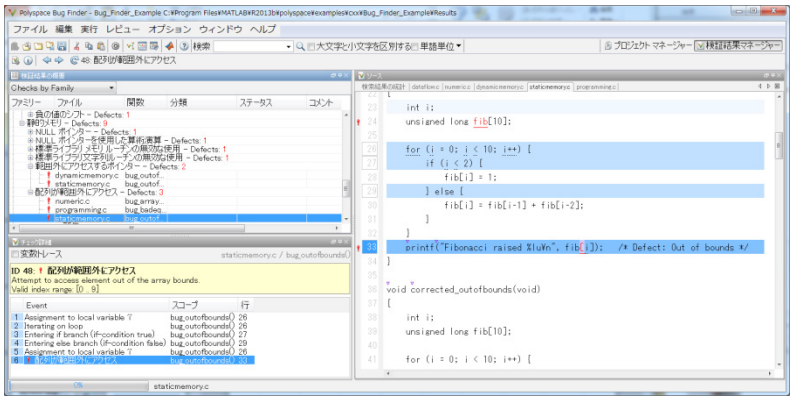

図2: Polyspace Bug Finderの検証結果

Polyspace Code Proverによるコード証明

Polyspace Code Proverは実績豊富な形式手法である抽象解釈を使用します。この技法はソフトウェアコンパイル時の動的性質を検証し、一般の静的解析技術と動的検証のギャップを埋めます。抽象解釈はプログラムを実行せずに、プログラム内で全ての可能な振る舞い－全ての入力の組み合わせと実行シーケンス－を検査して、ランタイムエラーが発生する状況を明確にします。

抽象解釈について

抽象解釈はソフトウェアアプリケーション等の複雑な動的システムに対してルールを定義する幅広い数学的な定理を使用して解析します。プログラムの各状態を一般化して表して、操作させるルールを定義します。抽象解釈は数学的な抽象化だけではなく、抽象化を解釈します。プログラム状態を数学的に抽象化するため、コード内の全ての変数を徹底的に分析します。以前はこの解析に対応できる処理能力を容易に入手することはできませんでした。現在のアルゴリズムと処理能力の増加により、複雑な検証の課題に対して抽象解釈は現実的な解決案です。

ランタイムエラー検出に適用することで、抽象解釈は全ての危険性がある操作を検証し、各操作が正常、エラー、非到達か、未証明と自動的に判断します。エンジニアはテストの最も早期段階であるコンパイル時に抽象解釈を使用できます。（付録A、Bを参照）

抽象解釈によるコード証明のメリット

抽象解釈は高信頼性が求められる組み込みソフトウェアを確保するにあたって費用と効率が良い手法です。コードを証明することで以下の4つの利点があります

コード信頼性の確保

抽象解釈はコードを徹底的に検査することで、ランタイムエラーの検出だけではなく、コードの正確性も証明します。これは安全性が最も必要とするシステムには重要になります。一般のデバッグツールはバグを検出しますが、残るコードのロバスト製を確認しません。抽象解釈はコード内に障害がないことを識別することができ、ソフトウェア信頼性に関する曖昧な点を解決します。

効率と向上

抽象解釈はアプリケーションの動的性質を検証することで、ランタイムエラーが存在しないコードと信頼性の侵害に繋がるコードを識別できます。コードを実行する前にエラーを指摘することで、修正が最も容易な段階でエラーを消去して時間とコストを大きく削減します。

オーバーヘッドの削減

抽象解釈はコードを実行する必要がありませんので、テストケースを作成・実行するコストなく高精度の解析結果を得ることができます。

デバッグ作業の容易化

抽象解釈はエラーの症状だけではなく、エラーの発生点を指摘することでデバッグを容易化します。エラーの原因を分析するためのトレース作業や、再現するための時間を削除することができます。抽象解釈は徹底的な解析の上、リピート可能です。コード内の各操作は自動的に全ての入力の組み合わせを取り組んで指摘、解析、検査します。

まとめ

Polyspace Code Proverの静的解析は抽象解釈を用いてコードを検証します。Polyspace Bug Finderと組み合わせることで、開発の早期段階でバグ検出、コーディングルールチェックと、ランタイムエラーの証明を可能とします。この手法により、組み込みソフトウェアを最も高い品質と安全性で動作する高信頼性を確保します。

付録A：抽象解釈の適用

抽象解釈を理解するにあたって、2つの変数X、Yを使用するプログラムPを考えます。下記の操作を行います：

X = X / (X – Y)

この操作でランタイムエラーの確認をする際、操作で可能なエラーの可能性を識別します：

- X、Yは初期化されていない可能性
- X – Yでオーバーフロー、アンダーフローの可能性
- XとYが同じ値になり、ゼロ除算する可能性
- X / (X – Y)でオーバーフロー、アンダーフローの可能性

上記状況はランタイムエラーを発生させる例ですが、下記ではゼロ除算の可能性に集中します。

プログラムPでXとYの全て可能な値の組み合わせを表示することができます。図3の赤い線はゼロ除算を発生する（X、Y）の値の組み合わせを示します。

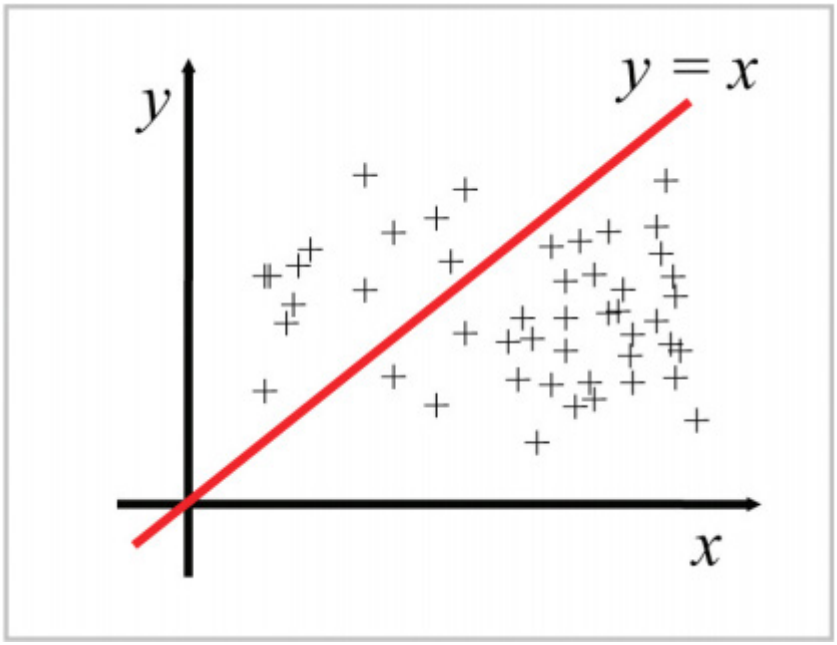

図3: プログラムPでXとYの全ての可能な値の組み合わせ

ゼロ除算を確認するための明白な方法として、すべての状態を列挙し、赤い線と重なるかを確認することです。一般のテスト手段はこの方法を利用しますが、本質的な限界があります。現実的なアプリケーションでは、多数の変数を使用しており、非常に多い状態が可能になります。全ての可能な状態を徹底的に確認することは不可能です。

全ての可能な状態を確認するようなしらみつぶしの計算と対照的に、抽象解釈は状態の集合を操作するための一般ルールを定義します。性質を証明するためにプログラムを抽象化します。

抽象化の1つの手法としてタイプ解析があります（図4）。このような抽象化はコンパイラ、リンカーや基本的な静的解析で使用されています。タイプ解析では、XとYの最小値・最大値を認識し、矩

形領域として表現します。矩形領域内にはXとYの可能な組み合わせを含むため、矩形領域の性質を証明することでプログラムに対して有効になります。赤い線と矩形領域が重ならない場合ゼロ除算の可能性はないと分かります。

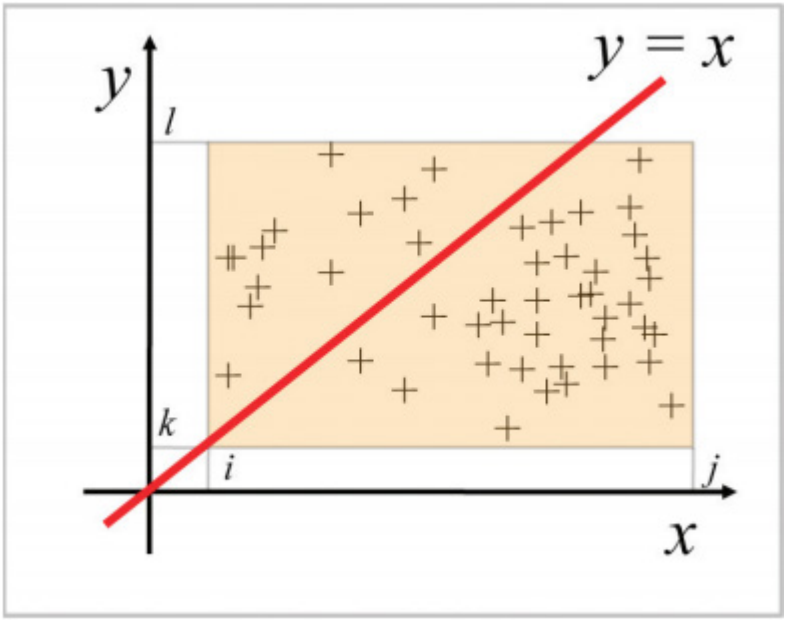

図4: タイプ解析

しかし、不正確な解析による結果がタイプ解析の欠点です。多数の誤検出警告が結果にあり、多くの場合、未読のままになります。

抽象解釈は矩形ではなく、より正確な多角形を定義します（図5）。制御構造（`if-else`文、`for`ループ、`while`ループ、スイッチ等）、手続き間操作（関数呼び出し）や、マルチタスク解析を考慮しながらデータ（X、Y）の関係性を整数格子、多面体の結合と、グレブナー基底を使用して表します。

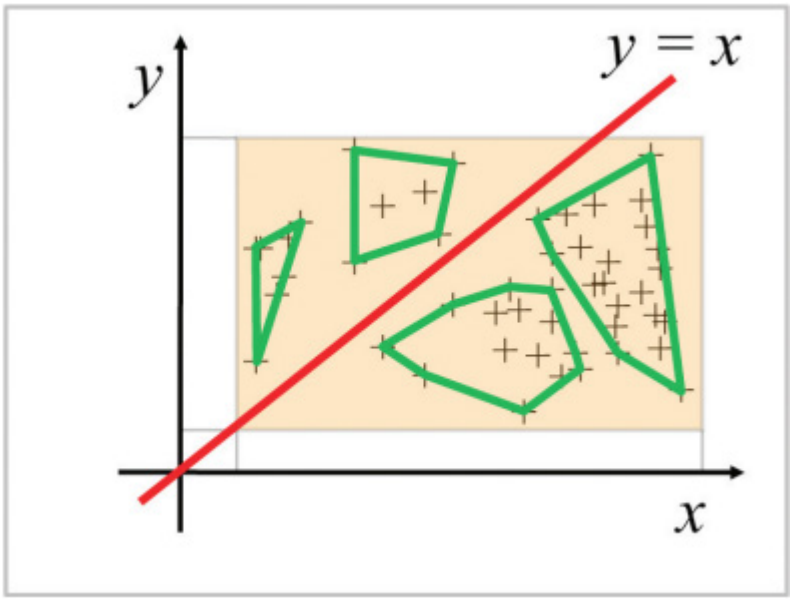

図5: 抽象解釈による多角形の定義

抽象解釈は、コンパイラや一般静的解析のようなデータ型と定数の関係だけの計算ではありません。プログラムの操作と計算対象の動作の関係性を派生し、コードを検査する際に利用します。

付録B: 抽象解釈の例

下記の例ではランタイムエラーを含むコードを示します。Polyspace Code Proverは抽象解釈により、このランタイムエラーを検出します。

制御構造解析: `for`ループ後の範囲外ポインターアクセス

```
10:   int ar[100];
11:   int *p = ar;
12:   int i;
13:   for (i = 0; i < 100; i++; p++)
14:     { *p = 0;}
15:   *p = 5;
```

上記例では、ポインターpは配列`ar`の先頭から始まる離散増加関数として抽象化できます。`for`ループが完了するとiは100であり、ポインターpも100に増加されています。配列インデックスの範囲は0から99のため、行15で範囲外となります。抽象解釈はこのコードに対して15行目にランタイムエラーが発生することを証明します。

制御構造解析：ネストされた`for`ループによる範囲外配列アクセス

```
20:   int ar[10];
21:   int i,j;
22:   for (i = 0; i < 10; i++)
23:   {
24:     for (j = 0; j < 10; j++)
25:     {
26:       ar[i - j] = i + j;
27:     }
28:   }
```

変数iとjは0から9に1刻みで増加します。配列`ar`のインデックスとなる操作i−jは負の値になります。抽象解釈はコードの行26に範囲外配列アクセスを指摘します。

上記例では、ランタイムエラーにより配列`ar`データの破損を発生します。破損データを使用する際のデバッグ作業は、抽象解釈を使用しない場合大変な工数が掛かる場合もあります。

手続き間解析：ゼロ除算

```
30:   void foo (int* depth)
31:   {
32:     float advance;
33:     *depth = *depth + 1;
34:     advance = 1.0/(float)(*depth);
35:     if (*depth < 50)
36:     foo (depth);
37:   }
38:
39:   void bug _ in _ recursive ()
40:   {
41:     int x;
42:     if (random _ int())
43:     {
44:       x = -4;
45:       foo ( &x );
46:     }
47:     else
48:     {
49:       x = 10;
50:       foo ( &x );
51:     }
52:   }
```

上記例では、変数depthはまず1増加する整数です。その後、advanceの計算処理で使用され、再帰的に関数fooで使用されます。行34でのゼロ除算の確認は、関数fooの引数の値を決定して*depthの値を把握するため、呼び出し間の解析が必要です。

関数bug _ in _ recursiveでは関数fooの呼び出しが つあります。行42のif分がfalseの場合、fooの引数が10になります（行50）。そのため、*depthは11から49の範囲の離散増加関数になります。行34の操作はゼロ除算の可能性はありません。

しかし、行42のif分がtrueの場合、fooの引数が−4になります。*depthは-3から49の範囲の離散増加関数になります。そのため、*depthが0のケースもあり、行34でゼロ除算を発生します。

単純なシンタックスチェックでは上記エラーは検出されません。抽象解釈は行45以外のコードは証明します。このように抽象解釈は呼び出し間解析を行い、正常な関数と危険性をもたらす関数を識

別します。上記エラーが修正されていない場合、ゼロ除算によりプログラムの停止が発生します。再帰的関数を含むエラーのデバッグ作業は多くの工数増加につながります。

下記の例ではランタイムエラーを含むコードを示します。Polyspace Code Proverは抽象解釈により、このランタイムエラーを検出します。

制御構造解析：forループ後の範囲外ポインターアクセス

```
10:   int ar[100];
11:   int *p = ar;
12:   int i;
13:   for (i = 0; i < 100; i++; p++)
14:     { *p = 0;}
15:   *p = 5;
```

上記例では、ポインターpは配列arの先頭から始まる離散増加関数として抽象化できます。forループが完了するとiは100であり、ポインターpも100に増加されています。配列インデックスの範囲は0から99のため、行15で範囲外となります。抽象解釈はこのコードに対して15行目にランタイムエラーが発生することを証明します。

制御構造解析：ネストされたforループによる範囲外配列アクセス

```
20:   int ar[10];
21:   int i,j;
22:   for (i = 0; i < 10; i++)
23:   {
24:     for (j = 0; j < 10; j++)
25:     {
26:       ar[i - j] = i + j;
27:     }
28:   }
```

変数iとjは0から9に1刻みで増加します。配列arのインデックスとなる操作i−jは負の値になります。抽象解釈はコードの行26に範囲外配列アクセスを指摘します。

上記例では、ランタイムエラーにより配列arデータの破損を発生します。破損データを使用する際のデバッグ作業は、抽象解釈を使用しない場合大変な工数が掛かる場合もあります。

手続き間解析：ゼロ除算

```
30:  void foo (int* depth)
31:  {
32:    float advance;
33:    *depth = *depth + 1;
34:    advance = 1.0/(float)(*depth);
35:    if (*depth < 50)
36:    foo (depth);
37:  }
38:
39:  void bug _ in _ recursive ()
40:  {
41:    int x;
42:    if (random _ int())
43:    {
44:      x = -4;
45:      foo ( &x );
46:    }
47:    else
48:    {
49:      x = 10;
50:      foo ( &x );
51:    }
52:  }
```

上記例では、変数depthはまず1増加する整数です。その後、advanceの計算処理で使用され、再帰的に関数fooで使用されます。行34でのゼロ除算の確認は、関数fooの引数の値を決定して*depthの値を把握するため、呼び出し間の解析が必要です。

関数bug _ in _ recursiveでは関数fooの呼び出しが2つあります。行42のif分がfalseの場合、fooの引数が10になります（行50）。そのため、*depthは11から49の範囲の離散増加関数になります。行34の操作はゼロ除算の可能性はありません。

しかし、行42のif分がtrueの場合、fooの引数が−4になります。*depthは-3から49の範囲の離散増加関数になります。そのため、*depthが0のケースもあり、行34でゼロ除算を発生します。

単純なシンタックスチェックでは上記エラーは検出されません。抽象解釈は行45以外のコードは証明します。このように抽象解釈は呼び出し間解析を行い、正常な関数と危険性をもたらす関数を識別します。上記エラーが修正されていない場合、ゼロ除算によりプログラムの停止が発生します。再帰的関数を含むエラーのデバッグ作業は多くの工数増加につながります。

92980v00 03/16