CTO グウィン・フィッシャー | ホワイトペーパー | 2009 年 7 月

Klocwork Insight 解析エンジン

正確でスケーラブルな全プログラム解析

自動ソース コード解析は、ソースコードが弱点とする分野を特定、対処するために設計されています。このような弱点としては、セキュリティの脆弱性、ロジックエラー、実装の不具合、並列処理違反、まれな境界条件、その他多数の種類の問題を起こしうるコードが挙げられます。関連する研究の分野には静的解析の技術があり、数十年にわたって主として Lint ツールという形態で使用されてきました。現代の静的解析には、悪名高いかつての静的分析と共通する部分はほとんどありません。現在のツールは、さまざまな技術を使用して全プログラム解析を実行し、この技術の領域に新しいレベルの拡張性と精度をもたらしています。本書では、この目的を達成するための Klocwork® のアプローチについて説明します。

自動ソース コード解析のプロセスには、提供されたコードの豊かな表現またはモデルの構築、さらにそのモデルを通じて可能なあらゆる実行パスのシミュレーション、データ オブジェクトが作成、使用、破棄される方法と場所とを組み合わせた、これらのパスに対するロジック フローの策定が含まれます。

コードパスの予測とデータ オブジェクトのマッピングが可能になれば、攻撃対象となる脆弱性、実行エラー、またはランタイムのデータ破壊を生じさせる可能性のある変則的な条件を見つけることができます。

全プログラム解析を構成するすべての要件の組み合わせですから、極端に大きく複雑なコード ベースのコンテキストでは、明らかに非常に複雑なアクティビティとなります。また、実際に、完全なソリューション（堅苦しく言えば、自明でない状態のスペースが決定不能なプログラムにおいてあらゆる不具合を検出するという目標 1）はないように思われます。

これらの類似した問題の中で、全プログラムの静的解析とは、モデルが現実には解析対象の実際のプログラムの近似値であるとしても、識別された不具合に関して、優れた検出精度を提供できるモデル、またはモデルの組み合わせを発見するということです。

技術

全プログラムの静的解析に使用される方法は、いまだに博士課程の研究対象であり、ホワイトペーパーの文脈で完全に説明できる内容ではありません。ここでは問題と考えられる解決策の概要を説明し、この領域と関連する問題の概要を説明します。

[1] http://en.wikipedia.org/wiki/Halting_problem および http://en.wikipedia.org/wiki/Rice's_theorem を参照

本書によって、読者は全プログラム解析の環境を構成する内容、解析に不正確さが生じる可能性のある点、以下の技術を組み合わせて、正確でスケーラブルなソフトウェア ソース コードの解析を生成する、Klocwork Insight を使用した、この問題の領域への独自のアプローチ方法について理解できます。

- ビルド プロセスの知識 - 全プログラム解析の基礎
- コード コンパイルと構文の解析 - 解析に必要なモデルの生成
- 制御フロー グラフでのデータフロー解析 - 最新のソース コード解析のコア技術
- 記号論理学 - ランタイム動作推論のためのスケーラブルなアプローチ

全プログラムの静的解析は、高品質な最新のソース コード解析技術をかつての技術と差別化するための技術の集合と言えるものであり、効果的な全プログラムの静的解析に確実な方法はありません。

ビルドプロセスの理解

あらゆる種類の正確な全プログラム解析で最も重要な点は、システムがネイティブのビルド環境を理解し、正確に再現できる能力に依存します。つまり、ネイティブのツールチェイン（コンパイラ、リンカーなど）を使用して、プログラムが実行できるようにどう構築されているかを解釈できる必要があるということです。ネイティブのビルドがメーカーのツールなどの標準的なツールに依存する場合でも、特定のソフトウェアシステムの構築のためだけの、完全なカスタムビルド環境に依存する場合でも、ビルドモニタ（ビルドを監視するプログラム）は同じようにビルド中の処理を理解できる必要があります。

ビルドから、モニタはビルドターゲットの正確なリストと各ターゲットの設定を決定し、解析がビルド自体と同じ方法で実行されるようにする必要があります。これは、ファイルがどのようにコンパイルされ、リンクされたかとともに、コード生成、ファイルのコピーなど、結果に影響するその他の操作についても理解するということです。当然ながら、エミュレーションを正確に行うため、解析エンジンが理解できるように、各コンパイラが言語を変更するため、ビルドプロセスで最も複雑な部分は、コンパイラの動作が中心となります。

たとえば、使用されているツール チェインが AltiVec[2] 命令セット（3D グラフィックス処理を目的とした C/C++ ベクトル数学拡張機能のセット）をサポートしているとモニタが判断できれば、動作しているセマンティックスに関して、評価を行うよう解析エンジンに指示することができます。しかしながら、モニタがこの機能の存在に気付かない、または誤った解釈をした場合、ソースコードでこのような特殊な命令に遭遇するとすぐ、実行される解析が無意味になります。

要するに、正確なビルドモニタリングが行われていないと、解析はシステムのネイティブ モデルを使用して実行され、おそらく、不正確なコンパイラのエミュレーションや言語仕様、または場所を含む不正確なファイル、または不完全なリンク ターゲットを使用して行われることになります。これらはすべて解析に大きな影響を与え、解析に帰結するほぼすべての事象の大部分が無意味になります。

Klocwork は、各々特定のビルド環境または言語を対象とした、さまざまなツールでビルドモニタリングの問題に対処しています。ただし、最も汎用的なツールである *kwinject* が大多数で使用されます。名前が示すとおり、単純にネイティブのビルドプロセスに注入することで、すべてのコマンドの起動を取得してログに記録します。インジェクションは実行中のネイティブのビルドによって実行されるため、取得されたビルドの形態は、関連するすべてのツール チェイン構成情報とともにコンパイルとリンクターゲットに関して完全なものになります。この情報を入手することで、Klocwork の解析エンジンを使用して、忠実さを損なわずに全プログラムを解析できます。

コードコンパイルと構文の解析

全プログラムの解析は、多くの意味で、ネイティブのビルドプロセスで実行されるステップに類似した、いくつかの主要なステップで進行します。中間表現を作成するコンパイル段階、中間表現を「最適化」する解析段階、最終表現を作成するリンケージ段階があります。当然ながら、違いは解析に使用される表現モデルがオブジェクトコードやバイナリではなく、不具合の発見をサポートするデータモデルであるという点です。

コードコンパイル

解析の最初のステップとして、明らかにコードのコンパイルは正しい結果を得るために最も重要です。その目標はマクロ定義、インクルードファイル、言語仕様などに関して同じコンパイラスイッチを使用して、ネイティブのビルドと全く同じ方法でプログラムを構成するさまざまなソースモジュールをビルド（解析のために）することであるため、これは完全にビルドモニタの出力によって決定されます。

Klocwork は、さまざまな言語（C/C++、C# および Java）に複数の異なったコンパイラを用意しています。これらの各ツールは、入力ソースコードを制御フローおよびデータフロー解析をサポートする中間表現に変換します。その間、ソースの抽象構文ツリー（AST）の初期のロスレス変換の作成も行います。

---

[2] http://en.wikipedia.org/wiki/AltiVec を参照

以下で説明するとおり、ソースコードの様々な表現形式が、さまざまな解析エンジンで使用されます。あるものはコーディングガイドラインに関連性の高い不具合を発見し、別のものはランタイムの障害に関連性の高い不具合を発見します。これにより、さまざまな種類の不具合を検索します。

「ビルドプロセス」の説明のとおり、コンパイル段階できわめて重要な側面の 1 つですが説明されていないことが多く、解析でエラーの原因になることが多いのが、プラットフォームセマンティックスの伝播（プラットフォーム独自の動作の理解）です。つまり、静的解析（オブジェクト コードを作成するネイティブのコンパイラではなく）に使用されるコンパイラが、付加的にコードコンパイルで意図するプラットフォームが使用するセマンティックスを理解し、AST を使用して伝播も行うということです。これは、たとえば、スレッディングモデル、メモリ割り当てモデル、一般的なプラットフォーム API セットなどに及ぶ場合があります。このステップを実行しなかったり、このステップを不正確に導入したりすると、作成されるコードが無効なモデルとなり、潜在的に naïve な解析が下流工程で実行される可能性があります。

構文解析

実行が最も簡単な解析の形態は、ソースの構文と表面のセマンティックス（明示的なタイプの伝播、組み込み型の拡張のサポートなど）に純粋に関連します。このプロセス中、AST は制限される使用法のパターンに関してスキャン、またはクエリされます。AST が開発者のソースのロスレス変換である場合、AST のすべての検査または検証は、言語仕様によって命令される厳格な階層型 / エンティティスキーマのコンテキスト内で生じます。たとえば、図 1 に示される C コードスニペットと付随する AST（理想化されています - 実際の AST はより複雑です）を見てみましょう。

```
void* foo(int i, int j)     |      FunctionDefinition = {
                            |        Name = "foo",
                            |        Type = BuiltinType(PointerVoid),
                            |        Parameters = {
                            |          Parameter = {
                            |            Type = BuiltinType(ValueInteger),
                            |            Name = "i"
                            |          }
                            |          Parameter = {
                            |            Type = BuiltinType(ValueInteger),
                            |            Name = "j"
                            |          }
                            |        }
{                           |        Body = {
  void* ptr = NULL;         |          Declaration = {
                            |            Type = BuiltinType(PointerVoid),
                            |            Name = "ptr",
                            |            Initialization = {
                            |              CastExpression = {
                            |                Type  = BuiltinType(PointerVoid),
                            |                Value = LiteralExpression(0)
                            |              }
                            |            }
                            |          }
   if( i < j )              |          IfStatement = {
                            |            Condition = {
                            |              RelationalExpression = {
                            |                Check = BuiltinOperator(LessThan),
                            |                Left  = IdExpression("i"),
                            |                Right = IdExpression("j")
                            |              }
                            |            }
                            |            TrueBranch = {
        ptr = malloc(32);   |              AssignmentExpression = {
                            |                Left  = IdExpression("ptr"),
                            |                Right = {
                            |                  FunctionExpression = {
                            |                    Name = "malloc",
                            |                    Arguments = {
                            |                      Argument = {
                            |                        Type = BuiltinType(ValueInteger),
                            |                        Value = LiteralExpression(32)
                            |                      }
                            |                    }
                            |                  }
                            |                }
                            |              }
                            |            }
```

図 1 | C コード スニペットと付随する AST の例

すぐにわかるとおり、言語自体は、言語要素を既知の構造とトークンに変換するように AST 内で形式化されています。このような構造を配置することで、制御される必要のある状況について、AST に対してどのようなチェックを実行できるかがすぐにわかります。

たとえば、特定の組み込みデバイスが従来のヒープ構造が使用できないような形で、あるいは少なくとも厳格に制御されるよ

```
FunctionExpression = {
  Name="malloc"
}
```

うに、実行中の環境にメモリ管理の制約を課すと仮定します。このような環境で、標準的な C メモリ管理ライブラリへの呼び出しを許可するのは、全く非生産的です。図 1 に示されるような理想化された AST 構造を使用すると、次の要素をエラー メッセージと関連付ける、このシナリオに対するチェックを構築できます。

つまり、周囲の言語の制約や問題になるメモリ管理関数に渡すためにコードがどの引数を選択するかは気にせず、単に関数呼び出しが発生するごとにフラグを付けたいということです。

同じことを単にグレップを使用してできないのでしょうか。当然ながら、AST 検証にはこの簡単な例より、はるかに豊富な機能があります。標準的な AST チェックで、このような「好ましい」コーディング ガイドラインは以下の内容に及びます。

- スタック上で値ごとに大型の構造型を渡すように試行する
- 条件における割り当て（またはその他の副作用）
- 論理演算子をバイナリ演算子と不注意に置き換える（「||」と「|」など）
- ワイド キャラクタ アプリケーションでナロー キャラクタ API を使用するなど、API の禁止事項
- 関数のリターンコードが無視された
- 戻り型の不一致
- 好ましいクラス構成（スコット メイヤーズ型のルール）
- その他多数

実際、オープンソースと営利団体両方で利用可能なコードチェッカーのコレクションを検討すると、このようなすべてのチェッカーの大多数は、AST に対して直接動作して、「そこにあるべきでない」何かを指摘します。

Klocwork の AST エンジン（KAST）には、幅広い組み込み型の構文チェッカーとサードパーティが独自の構文チェッカーを作成できる拡張メカニズムの両方が用意されています。この拡張メカニズムは、XPath から派生しており、豊富で階層的なコード検索パターンの定義をサポートしています。

たとえば、デフォルトの句を含まない C/C++ スイッチ文を検索する次の KAST 式を検討してみましょう。

```
//SwitchStmt
       [not Stmt::CompoundStmt/
        Stmts[*]::LabeledStmt /
        Label::DefaultLabel
     ]
```

このクエリ文は、スイッチ文のディスパッチャである CompoundStmt のさまざまなケースで構成されるラベル付きの文のコレクション内で DefaultLabel を含まないスイッチ文を検索します。

```
switch(foo())//SwitchStmt
{             //CompoundStmt
case 1:       //LabeledStmt(Label::CaseLabel)
default:      //LabeledStmt(Label::DefaultLabel)
}
```

若干複雑な例で、標準的な C/C++ コンパイラエラー（戻り値が void である関数から値を返す）を複製しますが、標準的なコンパイラに比べてはるかに機能の豊富な検証ルール セットの基礎を構成できます。

```
//FuncDef
      [DeclSpecs[*]::BuiltinType
      [@Spec=KTC_BUILTINTYPE_VOID]
      ]
      [FuncBody /
      Stmt::CompoundStmt /
      Stmts[*]::ReturnStmt
      ]
```

「void」型宣言があり、return 文を含む関数を検索するため、このクエリ文では、base 関数定義ノードに、2 つの制限が使用されています。

```
// FuncDef
void foo()                          // DeclSpecs
{                                   // FuncBody /CompoundStmt
    return 1;                       // ReturnStmt
}
```

要するに、AST に対して操作する構文解析は、以下が非常に優れているということです。

- 「拡張された」コンパイラの警告
- コーディングの規格とガイドライン
- 業界の「好ましい」ルール セット

ただし、構文解析はプログラムのランタイムの状態を理解する必要のあるエラーの検索に有効ではありません。この場合は、データフロー解析を使用します。

制御フローグラフでのデータフロー解析

前のセクションで見たとおり、構文と表面のセマンティックスでは、限られた不具合の発見しか行うことができません。実際、コーディング ガイドラインとスタイル ルールについては、AST が標準的なデータフロー関連のバグの複雑さを概算するだけで問題を発見できるため、これに反するルールを記述するのは困難であることから、ほとんどの場合、実際の不具合検出ではなく、構文解析が使用されるようになっています。

では、データフローとは何で、バグの発見にどう役立つのでしょうか。

データフロー解析とは、簡単に言えば、データ オブジェクトが制御フロー グラフ 3 の 1 つ以上のパスに従って作成、割り当て、使用、削除されるときに、データ オブジェクトのライフサイクルをモニタリングすることです。オブジェクトがライフサイクルのコンテキストに基づいて、不適切に使用されると、不具合が見られる場合があります。たとえば、削除された後、初期化される前、または問題のある値を割り当てられたことが判明する前に使用されたり、複数回削除されたり、まったく削除されないオブジェクトなどが挙げられます。これらの状況はすべてランタイムでの障害に至る可能性があり、すべて静的に発見できます。

次のセクションで検討するとおり、制御フロー グラフの構成中やデータフロー解析中に発生する不正確さは、高いノイズレベルや、ユーザの信頼に関する大きな問題に至る可能性があります。これは、最新のソース コード解析の中心であり、高水準の精度さでバグ検出を追求する製品で最も重要な点です。

Klocwork Insight のデータフロー解析エンジンを使用することによって、高価値の不具合を高い信号対雑音比で、つまり誤検出率に対して高い比率で実際の不具合を発見して報告します。このエンジンは、この解析を実施するために、本書の以降のセクションで見られるような複数の異なった技術を使用しています。

制御フローとプロシージャ間解析

前述のとおり、ソースコード解析ツールチェイン内のコンパイラが、データフロー解析に適したコードの中間表現への変換を行います。つまりこれは、中間表現でローカルの制御フローグラフ（つまり、その 1 つのコンパイル単位またはモジュール内のコードを通じて実行可能なパス）を反映する必要があることを意味しており、他のコンパイル単位への制御フロー変換をどこでどのように実行するかを明確にする必要があります。

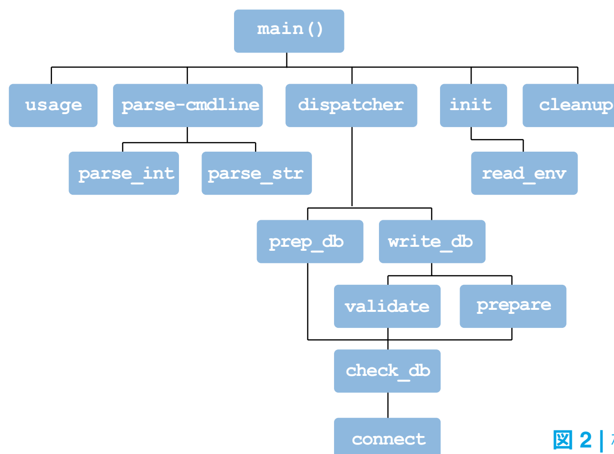

図 2 | 標準的なプログラムのコールグラフ

[3] See http://en.wikipedia.org/wiki/Data_flow_analysis

コンパイルされたモジュールの集合が存在する状況では、全プログラムのコールグラフを構成し、解析のために準備することができます。このコールグラフは、プログラムを構成するモジュールの解析に対して、プロシージャ間解析が適切な時間内に完了できるように命令するために使用されます。

したがって、コールグラフとして生成した内容は、プログラム実行中にすべての関数呼び出しを独自に反映し、非循環有向グラフ（DAG）と呼ばれる、よく知られたコンピュータ化学の構造物になります。

循環的な相互に関連する関数の呼び出しは複雑であることから、理解しやすくするため、ここでは検討しません。基本的には、DAG の解析は、一番下（リーフ）から開始し、一番上（ルート）に移行します。つまり、関数のローカル制御フローグラフの解析は DAG の「下位」にあたる、全ての呼び出される関数のコンテキスト情報が生成された状況で行われます。（図 3）。

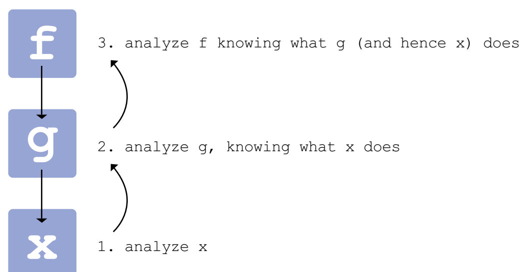

図 3 | DAG による行程

プロシージャ間解析は、コールグラフ内、またはコールグラフの複数のノードに関係する不具合の発見に関連します。プロシージャ内解析は、コールグラフの単一ノード内でのみ生じる不具合に関係します。このように、プロシージャ間解析をサポートするため、このコールグラフ処理への「ボトムアップ」アプローチである DAG は、呼び出す関数に関して生成された情報を使用して、各関数を独自に解析できることを意味します。この解析プロセスは、コールグラフで「上方に」進行するため、エンジンにはより多くのコンテキストが使用可能ですが、エンジンはまだ一度に単一の機能しか処理できません。このように解析結果に対して線型でアプローチすると、一般性をほとんど損なうことなく、優れたパフォーマンスが得られます。

当然ながら、これを機能させるための鍵は、情報がコールグラフの「低位」レベルから、「高位」レベルへとどのように伝播されているかです。

Klocwork Insight エンジンは、関数の動作を要約するメソッドを使用して、各関数が入力パラメータ、環境とどのように相互作用し、リターンコードや出力パラメータをどう生成するかに関する各関数の動作についての重要な事実を記録します。この情報は解析プロセスがコールグラフのリーフからルートへ順次処理するに従い、ナレッジベースに記録され、蓄積されます。ナレッジベースの個々の記録は、エンジンを呼び出す関数の解析で重要と考えられる、関数の 1 つの特定の動作または特性をエンコードします。

たとえば、エンジンには関数「f」の解析中に、パラメータとして渡された特定のバッファが正しく処理されているか（範囲内）いないか（範囲外）を把握するために十分な情報がない可能性があります。このため、バッファが実行した処理の範囲を詳述する動作記録をエンコードし、「f」を呼び出す関数「g」が解析されているとき、バッファがオーバーフローしているかアンダーフローしているかをその時点でエンジンが判断できるようにします。

```
void f(char* buffer, int x, int y) {
    if( x > y ) buffer[x - y] = 'x';
    else buffer[y - x] = 'y';
}
```

この関数の例に関するナレッジベースでは、2 つの異なった動作記録を使用して、2 つの異なった関数「f」の動作が記録され、最初の動作は（x > y）の場合にオーバーフロー（または潜在的なオーバーフロー）動作、2 番目の動作はその逆を示します。いずれの場合も、呼び出し側を解析する際に、範囲が不正かどうかについてエンジンが決定を下せるように、配列アクセスの下限と上限がエンコードされます。

```
void g() {
    char buff[32];
    f(buff, 64, 10);        // Illegal access
    f(buff, 10, 41);        // No problem
}
```

要するに、このナレッジベースの最終版は、全プログラムがどのように動作するかの集大成、つまり概算であり、部分的なコンポーネントの解析（通常、個々の開発者のワークフローの一部として、コンポーネントまたはサブシステムが解析されるが、解析実行のために全プログラムのコンテキストの利点を活用する場合）など、Klocwork 製品スイート全体で、その他多数の操作の背景として使用されます。

記号論理学

Wikipedia では記号論理学、つまり記号プログラム解析を次のように定義しています 4。「すべてのプログラムは文法規則を使用して記述されており、標準的な記号である文法トークンを使用します。記号は変数名、クラスまたはメソッド名、または文法内の予約語と同じように、オブジェクトやオブジェクトの背後にある動作モデルを記述します。このアプローチでは、低レベルのプログラム知識の基礎が作られます。オブジェクトの理解は、プログラム記号間で解釈を作成することによって達成されます」

Klocwork Insight エンジンでは、ソフトウェアの動作を推測するために、記号論理学の基礎を使用し、そこから全プログラム解析を生成します。

ソフトウェア動作の伝播

上述のとおり、Klocwork Insight エンジンではプレースホルダ（つまり記号）とプレースホルダ間の相互関係を使用して、関数の動作を定義します。この動作の推論は記号論理学の中核に基づいており、関数の構文の「記号」（通常は変数またはパラメータ）を使用して、ランタイムに記号と一致する値または一連の値を理解する必要なく、動作の推測に使用できます。

記号論理学を解析の形式モデルとして使用することによって、Klocwork Insight エンジンは値がトリアージに使用できるようになるまで、コールグラフ全体で動作について推論し、伝播することができます。適用された値が、ナレッジベースで取得された記号による関係に該当する場合、エンジンはコードによって不具合が生じるかどうかを判断できます。

値のない簡単な例：

値の伝播または線形等式に依存するエンジン（標準的な格子ベースの DFA エンジンなど）は、2 つの入力パラメータ間の関

```
void foo(int i, int j) {
    void* ptr = NULL;
    if( i < j )
        p = malloc(32);
    if( i < j )
        free(p);
}
```

係について推論することはできないため、メモリ リークの不具合を報告する側について推論します。Klocwork Insight エンジンは、テストされた条件によって生成された固有の公理を理解し、メモリが適切に扱われていることを「認識」できます。

同様に、もう少し複雑な例で、同じ推論がプロシージャ間に適用される場合を見てみましょう。

```
extern const int MAX_SIZE;
void* myAlloc(int size, int limit)
{
    return size < limit ? malloc(size) :
NULL;
}
void *f(int n)
{
    if (n <= 0) return NULL;
    void* p = myAlloc(n, MAX_SIZE);
    if ( n > MAX_SIZE) {
      LOG(“Buffer size exceeds limit”);
      return NULL;
    }
    memset(p, 0, n);
    return p;
}
void *g(int n)
{
    if (n <= 0) return NULL;
    void* p = myAlloc(n, MAX_SIZE);
    if ( n >= MAX_SIZE) {
      LOG(“Buffer size exceeds limit”);
      return NULL;
        }
      memset(p, 0, n);
      return p;
}
```

4 http://en.wikipedia.org/wiki/Symbolic_Program_Analysis を参照

ここには、有効性を判断するプロシージャ間の記号による関係に依存した同じ関数について、2 つの代替的な実装が示されています。関数「f」では、着信パラメータ「n」と定数「MAX_SIZE」の間に誤った比較演算子（>）が使用されているため、「n」が値「MAX_SIZE」を持つと、クラッシュ（NULL ポインタ デリファレンス）が生じる可能性があります。対照的に、関数「g」では、正しい比較演算子（>=）が使用されており、このクラッシュのシナリオは回避されます。これら 2 つの状況を判断するには、エンジンが着信パラメータに関して関数「myAlloc」の動作を正しく要約し、その動作を既知の値なしで解析できるように呼び出し側に伝播する必要があります（「n」と「MAX_SIZE」が同じであり、関数「f」が失敗し、関数「g」が成功する場合は、その判断のために「n」または「MAX_SIZE」の値を知る必要はありません）。

この記号論理学の中核によって、Klocwork Insight は他の技術では手に負えない、多数の複雑な状況で関数の動作を理解できます。もう 1 つ最後の例を見てみましょう。

```
extern int error_code();
extern int* get_some_data();

int get(int *&aToken) {
    aToken = NULL;
    int res = error_code();
    if( !(res & 0x3) ) return res;
    aToken = get_some_data();
    return 0;
}
int foo() {
    int* p;
    int rc = get(p);
    if( rc & 0x3 ) *p = 1;
}
```

ここでもナレッジベースは記号による関係の伝播に使用されます。ここではリターンコードと入出力パラメータ「aToken」との関係になります。関数のリターンコードがビットマスク 0x3 に一致しない場合、出力「aToken」はまだ NULL です。ただし、一致する場合は、出力「aToken」には（おそらく）関数「get_some_data」への呼び出しから取得された有効な値があります。

ここで、Klocwork Insight エンジンの推論は記号の相互関係の推論だけでなく、条件ベースの公理（制約の利用可能な範囲が相当あるが、この例ではビットマスク）の形態で制約を適用し、最終解析ができる限り正確になるようにします。

以下はこの種の解析が実行されている「実際の」例です。最近の Firefox のブランチから取得したこのコードでは、ビットマップの制約が活用され、NULL ポインタがデリファレンスされて誤検知が報告される可能性が打破されます。

そのビットマスクの制約では、呼び出し側の関数で行われるチェックと明示的に競合するため、エンジンが「nsnull」戻り値を考えられるリターンコードのリストから正確に削除することを確認します。

```
nsRect
nsIFrame::GetOverflowRect() const
{
  // Note that in some cases the overflow area might not have been
  // updated (yet) to reflect any outline set on the frame or the area
  // of child frames. That’s OK because any reflow that updates these
  // areas will invalidate the appropriate area, so any (mis)uses of
  // this method will be fixed up.

  if (GetStateBits() & NS_FRAME_OUTSIDE_CHILDREN)
    return *const_cast<nsIFrame*>(this)->GetOverflowAreaProperty(PR_FALSE);
    // NOTE this won’t return accurate info if the overflow rect was updated
    // but the mRect hasn’t been set yet!
  return nsRect(nsPoint(0, 0), GetSize());
}
nsRect*
nsIFrame::GetOverflowAreaProperty(PRBool aCreateIfNecessary)
 {
  if (!((GetStateBits() & NS_FRAME_OUTSIDE_CHILDREN) || aCreateIfNecessary))
    return nsnull;
  ...
}
```

実行されないパスの排除

正確な解析を実行し、正確な関数の動作を伝播する鍵の 1 つは、解析対象の関数によって、有効なコードパスと無効なコードパスを正しく識別することです。これができない場合、関数のコードによって明示的に拒否される病的な動作を判断して、偽陽性がコールグラフのあらゆるレベルで報告されるように、動作を伝播します。

Klocwork Insight エンジンはこの問題の領域にも記号論理学を適用し、値で決定されているかどうかにかかわらず、コードで発見された公理と相互関係に基づいて収集するパスを識別できます。有効なパスのコレクションがあれば、不具合を生じさせる（または潜在的に不具合を生じさせる）状況をトリアージして、既知の有効なパスで生じていることをすばやく確認できます。

では、次の例について検討してみましょう。

```
#define MAX 32

extern char* get_some_data();

void foo(int i, int j) {
    char* p = NULL;
    if( i < MAX )
        if( j < MAX )
            p = get_some_data();
        else bar();
    else bar();
    do_something();
    if( j < MAX )
        if( i < MAX )
            *p = 32;
        else bar();
    else bar();
}
```

ここで判断できる必要があるのは、NULL 値を割り当てている「p」から、デリファレンスされて（書き込みされて）いる「p」の間に有効なコードパスがあるかどうか、あるいは有効なパスのみが「get_some_data」への呼び出しに基づいて割り当てられている「p」に帰結しているかです。

この関数を検討する場合、フローチャート（図 4）を使用すると、決定点がわかりやすくなります。図 5 には、この関数によるいくつかの無効なコードパスが示されています。

- 最初のパスでは、「j」が条件「< MAX」を満たしているかどうかについて、矛盾する決定が行われています。
- 同様に、2 番目のパスでは、「I」が条件「< MAX」を満たしているかどうかについて、矛盾する決定が行われています。

いずれの場合も、開発者は正しい条件が満たされた場合のみ（つまり、「i」と「j」が常に「< MAX」であること）コードが動作するように明示的に編成しているため、コードパスは無効です。Klocwork Insight エンジンは両方の条件について、相互関係を正しく識別し、正しいコードパスのみが選択されるように、これらの定理をチェックポイント間に伝播できます。

まとめ

Klocwork Insight 解析エンジンは、それ自体が実際には何を表しているのかを理解する必要なく（計算により）、変数間の相互関係を推論できるため、誤検知が削減され、有効な問題の識別が向上します。完全に統合されたビルドを解析しているか、コード チェック前のローカルプロジェクトを解析しているかにかかわらず、Klocworko Insight はあらゆるコードパスに対応する、4 方向からのアプローチで精度と対応範囲で最適のバランスを提供しています。

- ビルド プロセスの知識
- コード コンパイルと構文の解析
- 制御フロー グラフでのデータフロー解析
- 記号論理学

このような独自の技術の組み合わせで、Klocwork Insight は、C/C++、Java、C# の開発者に高価値、低ノイズの解析を提供しています。

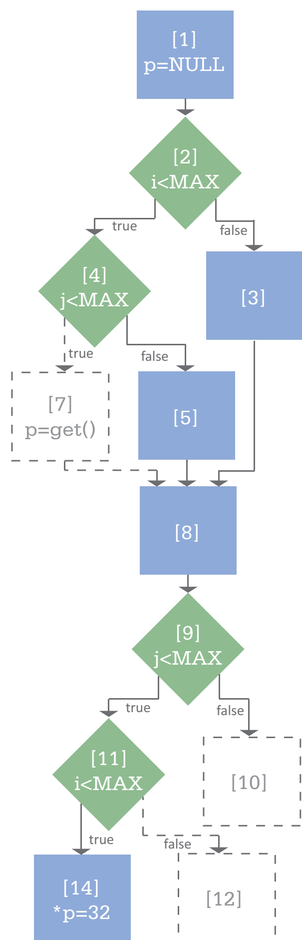

図 4 | 関数のフロー チャート

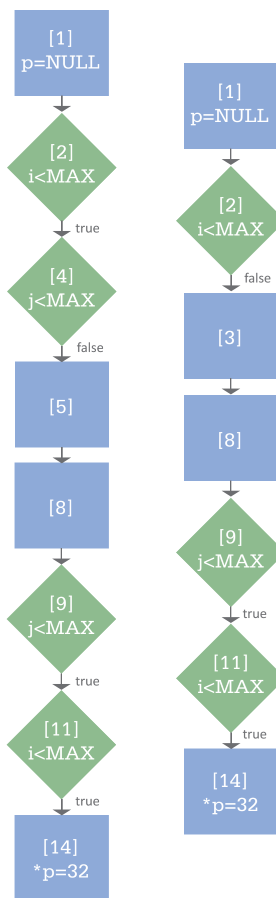

図 5 | 関数のフロー チャート

著者の紹介

グウィン・フィッシャーは Klocwork の CTO であり、同社の技術的な方向性と戦略を担っています。海外での 20 年近い技術経験があり、ソフトウェア開発に関する優れたビジョンと経験、洞察をあわせ持っています。形式文法と数理言語学の分野での経験を持つフィッシャーは、キャリアの大半を検索と自然言語の開発・研究に費やし、Hummingbird、Fulcrum Technologies、PC DOCS、LumaPath などの企業で上級管理職を務めてきました。Klocwork に入社してからは、従来からの関心事であるコンパイラ理論に重点を置き、静的ソース コード分析を次のレベルへと進化させるため、開発者ならではの経験と知識を注ぎ込んでいます。

Klocwork について

Klocwork® はデベロッパーがより安全で信頼性の高いソフトウェアを作成するのに役立ちます。弊社のツールはソースコードをオンザフライで解析し、ピアコードレビューを簡素化し、複雑なソフトウェアの寿命を延ばします。モバイル機器、家庭用電化製品、医療技術、通信、自動車、軍事、航空宇宙部門の最大ブランドを含む 1100 社を超えるカスタマーが、既に Klocwork を自社のソフトウェア開発プロセスの一部に組み込んでいます。数多くのソフトウェア開発者、設計者、そして開発マネージャーが弊社ツールを日々活用して、生産性を高めると同時によりよいソフトウェアの開発を行っています。詳細に関しては、www.klocwork.com または info@klocwork.com にて Klocwork までお問い合わせください。

米国：
15 New England Executive Park
Burlington, MA 01803

カナダ：
30 Edgewater Street, Suite 114
Ottawa, ON K2L 1V

t: 1.866.556.2967
f: 613.836.9088
WWW.KLOCWORK.COM

klocwork®

© Klocwork Inc. All rights reserved.