Vol.2018-SE-198 No.18
2018/3/9

Mollis: オープン実装に基づいた静的解析ツールの提案と実装

大島 健太　福田 浩章

概要：近年のソフトウェア開発において, プログラムのセキュリティや生産性向上の為に静的コード解析ツールが開発され, 成果を上げている. しかし, ソフトウェア開発チームや個人の開発者の解析ツールに対する要求は多種多様であり, 単一の解析ツールのみで様々な要求に対応することは困難である. また, 解析ツールで行う処理はコンパイラと同様に複雑であり, 一般に開発者による処理の変更は考慮されていない (ブラックボックス). その為, 開発者が自身の要求に合わせて解析ツールを一から設計し実装するためには, ブラックボックスの中身を理解しなければならず, 容易ではない. この問題を解決するために, 警告を表示するプログラムのパターンを開発者が設定可能な解析ツールや, プラグインによる機能の追加が可能な統合開発環境が開発されている. これらを用いる事で要求通りの環境を整えることが容易となるが, これらの解析ツールは抽象構文木の作成等, 解析ツール内部で行われる詳細な処理は, カスタマイズが不可能, もしくは制限されている場合がある. そのため, 特定クラスに限定して解析を行いたいという要求や, 抽象構文木の形を変えたいといった要求等, 解析ツール内部の処理に変更を加える必要がある場合, 前述した解析ツールでは不十分である,
そこで本研究では， 開発者の要求に応じて柔軟に機能を変更できるよう, 解析に必要な様々な処理がカスタマイズ可能な解析ツール, Mollis の提案と実装を行う．Mollis では，オープン実装の思想に則り，開発者に内部実装を変更できる API を提供する．開発者は, Mollis が提供している API を利用して解析処理のカスタマイズを行う事ができる．また，本論文では具体的なカスタマイズ例，および解析に要する時間を計測することで，Mollis の柔軟性が十分であり, 現実的な時間で解析できることを示す．

Mollis: Proposal and Implementation of Open Implemented Static Analyzer

Oshima Kenta　Fukuda Hiroaki

Abstract: Static analyzer used to improve security and productivity of software. However, there is a wide variety request for analyzer, and it is hard to resolve in single analyzer. Static analysis is as complicated as compiler, general developers can't understand all processes of analysis. However, in the case of developers must change analyze process, developers must understand all processes. Therefore, it is likely that developers feel hard to customize of static analysis. On the other hand there are some solutions, for example, plug-in system is useful for customize. However, that solutions also not work to developers that want to customize the detail of process. This paper propose Mollis, that is static analyzer based on open implementation. In Mollis, according to the idea of open implementation, developers can change internal implementation by using provided API. Flexibility and analyze speed are viewpoints for evaluating, to evaluate the effectiveness of Mollis, we evaluated them.

1. はじめに

近年のソフトウェアは, 大規模化や高機能化に伴い, ソフトウェアのソースコード行数も増大しており, 目視等の手作業でソースコードの安全性を評価することが困難になっている. そこで, ソースコードの安全性等を自動的に評価を行うことのできる静的解析ツールが使用され, 成果を挙げている [1]. しかし, ソフトウェアの開発現場では, 企業や

プロジェクトごとに, コーディング規約や重視すべき機能等が異なる可能性がある. そのため, ソフトウェアの開発者は, 開発効率の向上やソースコードの安全性を向上させるために, 自身の環境に応じて解析ツールを使い分けたり, 解析ツールの自作や解析処理のカスタマイズを行う必要がある. しかし, 解析ツールが行う処理はコンパイラと似ていて複雑であり, 環境の変化の度に解析ツールを一から自作して, 最適な解析環境を整える事は困難である. また, 解析ツールが解析処理を柔軟にカスタマイズできるような機構を開発者に提供していない限り, 解析処理のカスタマイズも行うことができない. さらに, 解析ツールを使い分ける場合でも, 変化の度にツールの扱い方を学習しなければならず, 開発の効率は低下してしまう. この問題を解決するために, 警告を表示するプログラムのパターンを開発者が設定可能な解析ツールや, プラグインによる機能の追加が可能な統合開発環境が開発されている. これらを用いる事で要求通りの環境を整えることが容易となるが, これらの解析ツールは抽象構文木の作成等, 解析ツール内部で行われる詳細な処理は, カスタマイズが不可能, もしくは制限されている場合がある. その為, 特定クラスに限定して解析を行いたいという要求や, 抽象構文木の形を変えたいといった要求等, 解析ツール内部の処理に変更を加える必要がある場合, 前述した解析ツールでは不十分である.

そこで本研究では, 開発者の要求に応じて柔軟に機能を変更できるよう, 解析に必要な様々な処理がカスタマイズ可能な解析ツール, Mollis の提案と実装を行う. Mollis では，オープン実装の思想に則り，開発者に内部実装を変更できる API を提供する. 開発者は, Mollis が提供している API を利用して解析処理のカスタマイズを行う事ができる. また，本論文では具体的なカスタマイズ例，および解析に要する時間を計測することで，Mollis の柔軟性が十分であり, 現実的な時間で解析できることを示す.

2. 関連技術

本節では, 既存の関連製品について紹介し, それらが持つカスタマイズ機能について述べる. また, 既存の関連製品における柔軟性の問題点について述べる.

2.1 FindBugs

FindBugs[2] は, Java プログラム用に設計された, オープンソースの静的解析ツールである. バグランクと呼ばれる指標等を用いて, 警告のフィルタリングを行う事ができる一方で, 解析処理のカスタマイズを行う事ができず, 柔軟性が低い.

2.1.1 バグランクによる分類

FindBugs が検出できるバグは, それぞれがバグランクと呼ばれる指標を保持している. バグランクは, バグの危険性を 1 から 20 の数値で表しており, 危険であるほど数値は低く設定される. バグランクは開発者がコードのどの箇所を修正すべきかを判断する材料であると同時に, フィルター機能を用いたバグの絞り込みに用いる要素でもある. フィルター機能を用いることで, 開発者が必要な情報だけを抽出して警告に出すことができるため, フィルターは一種のカスタマイズと考えることができる.

2.1.2 FindBugs の問題点

FindBugs は, フィルター機能を用いる事で警告の絞り込みが出来るが, 新たにバグを表示するパターンの定義を行ったり, 独自の解析手法でバグの検出を行うことはできない. そのため, 解析処理そのものをカスタマイズしたい場合は, FindBugs が提供している機能だけでは不十分であり, 柔軟性に欠ける.

2.2 Eclipse

Eclipse[3] は, IBM によって開発されたオープンソースの統合開発環境であり, Java をはじめとした様々なプログラミング言語に対応している. Eclipse の特徴として, プラグイン機能が挙げられる. プラグインとは, ソフトウェアの機能を拡張するために差し替えることが可能なプログラムであり, Eclipse のユーザは使用したい機能を持つプラグインを適用させることで Eclipse の機能を最適化する事が可能である.

2.2.1 Eclipse JDT

Java Development Tool(JDT) は, Eclipse で Java の解析等を行えるようにするプラグインであり, Eclipse 本体が, JDT を利用して動作している. JDT を利用することで, Eclipse プロジェクトで扱っている Java ファイルの情報にアクセスできるため, 開発者は自身の要望に合った解析を行うことができる. アクセスできる Java ファイルの情報は Java の文法を元に生成された抽象構文木であるため, 記号表の作成等, 抽象構文木の情報を必要とする処理は自由に行う事ができる.

2.2.2 JDT における問題点

JDT は Java の抽象構文木情報を提供しているが, 抽象構文木の前段階である解析木の情報についてはアクセスすることができず, 抽象構文木の作成をカスタマイズする事はできない. そのため, 開発者が抽象構文木の構造を変更したり, 解析木を走査する中で何かアクションを行いたい場合は JDT の機能だけでは不十分であり, こちらも柔軟性が高いとは言えない.

3. アプローチ

本節では, オープン実装 [4] による柔軟性向上の提案と, API を用いたオープン実装の実現手法について述べる.

3.1 オープン実装

オープン実装とは, ソフトウェアの設計思想, 設計手法の

ひとつであり, ユーザにソフトウェア内部の処理を変更できるように予めインタフェースを提供して実装することである. 近年のコンパイラや静的解析ツール等は, 内部の処理が複雑化しており, 詳細まで理解する事が難しくなっている. そのため, 解析処理はブラックボックス化しており, ソフトウェアに小さな変更を加える場合でも, ソースコードのどの箇所をどのように変更するべきか知るために大きな労力と時間が掛かる. しかし, 適切なインタフェースを備えたソフトウェアであれば, 内部の処理について十分な知識を持っていない開発者でも, 容易に機能の変更を行うことができる. 図 1 は, オープン実装の考え方を示している. このように, オープン実装では, 開発者とソフトウェアの処理 (ブラックボックス) との間にインタフェースを介在させることによって, 開発者の負担を減少できるというメリットがある.

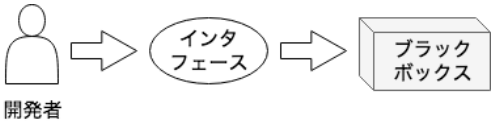

図 1 オープン実装の概要

3.2 API の提供

本研究では, 開発者に API を提供することでオープン実装を実現する. Mollis ではプログラムの解析を, 抽象構文木の作成, 記号表の作成と解決, コールグラフの作成と解決, 以上の 5 つのステップに分割し, それぞれのステップでデフォルトの解析処理と, カスタマイズに使用する API を提供する. これは, 解析処理を種類ごとに分割することによって, 開発者が自身の要望を実現させるために, どのステップの処理を変更すればよいか判別しやすくし, カスタマイズに掛かる手間を軽減させる目的がある.

例えば, 自身の書いたソースコードが, 所属している企業やプロジェクトのコーディング規約に則っているかを, 解析ツールを用いて検査したいという要望があると仮定する. プロジェクトが変化すればコーディング規約も変化する可能性があり, 規約の変化にあわせて解析処理も変化させる必要がある. 変数名やメソッド名の検査については, 抽象構文木を作成する処理の中で行うことができるため, Mollis を利用する開発者は, 1 つ目のステップのみカスタマイズすれば良く, 簡単に解析処理のカスタマイズを行うことができる.

3.3 解析処理

Mollis における解析処理は, 抽象構文木の作成, 記号表の作成と参照, コールグラフの作成と参照の 5 つのステップに分けて行われる. 以下では, それぞれのステップで行っている解析処理について述べ, 各ステップで行うことの出来るカスタマイズについて述べる.

3.3.1 抽象構文木の構築

抽象構文木は, 一般に情報の密度を向上させ, 走査をしやすくするに生成, 利用される. 具体的には, 解析木には適応した構文規則名や文末のセミコロンなど, 静的解析を行う上では不要な情報が多く含まれている. そのため, 不要な情報を消去して木を簡潔にし, 木の走査を簡単にする必要がある. 図 2 は, 解析木を抽象構文木へと書き換える処理を模したものだが, 抽象構文木は記号表の作成にもコールグラフの作成にも利用する為, 木の書き換えは 5 つのステップで最初に行われる.

Mollis では, 構文解析に ANTLR[5] が生成した構文解析器を利用しており, 構文解析器が構文規則ごとに行うアクションを変更して抽象構文木を作成する. また, 構文規則毎に行われるアクションは, Mollis を使用する開発者もカスタマイズすることができる. そのため, 3.2 で述べたようなコーディングスタイルのチェック等, プログラムの字面を検査に使用する場合は, このステップでのカスタマイズが有効である.

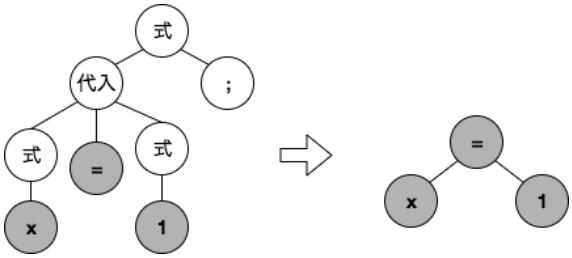

図 2 抽象構文木の構築

3.3.2 記号表の作成と参照

記号表は, ソースコードで定義された記号を記録する役割を担っている. 記号とは変数やオブジェクト, メソッド等の様々な情報の事を指す. 記号表に新たな記号を登録する処理は, 抽象構文木を走査する中で記号の宣言がされている箇所に到達した時に行う. また, 記号を登録する場合は, 記号が属しているスコープの情報と合わせて登録する必要がある. この記号表の作成は Mollis における第 2 ステップにあたる. 図 3 は, 記号表の作成の段階で生成するスコープ木を模したものである. 図 3 は, クラス Clazz にメソッド f, g が存在する事を示している. また, 内側のスコープから外側のスコープの記号は参照できるが逆は行う事ができない事を示すために, 矢印が親に向かって伸びている.

記号表の作成が完了すると, プログラム中に出現する記号の解決を行うことができる. 例えば, "sum = x + y;"というプログラムを解釈するためには, x と y の記号を解決する必要があり, そのために第 3 ステップである記号表の参照が用意されている. これら第 2, 第 3 ステップについてもオープン実装する事で, 記号の登録と解決を自由にカスタマイ

ズができるようになる. 例えば, 一度も解決が実行されない記号はプログラム中で使用されないことを示すため，それらを一覧で表示する機能を実現することで, プログラムの不要な文を消去することができ, プログラムの可読性や保守性の向上を図ることができる. Eclipse 等の一般的な解析ツールでは, 使用されていない変数やメソッドには GUI 上で波線が引かれ, 不要であれば消去するべきと削除を推奨するような手法が一般的だが, この手法ではソースコード上に散在する波線を全てチェックする必要があり, 修正に手間が掛かってしまう. Mollis のカスタマイズを行い, 使用されていない記号の一覧を, ファイル名や行番号と共に出力することで, 開発者は修正すべき箇所を簡単に発見することができる.

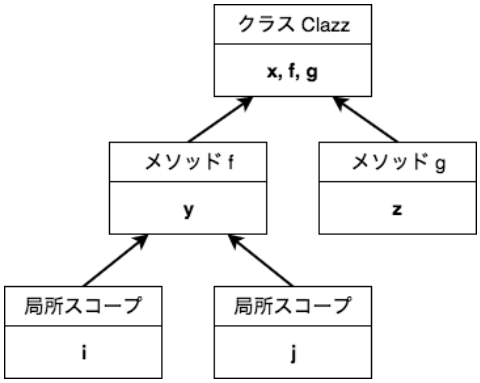

図 3 スコープ木の例

3.3.3 コールグラフの作成と参照

コールグラフは, ソースコード内のメソッド同士の呼び出し関係を表した有向グラフである. 図 4 は, コールグラフの一例であるが, このコールグラフから，main メソッドが method A, B, C を呼び出している事や, method E が method B, D に呼び出されていることがわかる. コールグラフにおける頂点はメソッド情報であり, 辺は呼び出し関係であるので, 抽象構文木を走査してメソッドの宣言を行っている箇所に到達した時新たな頂点ノードを作成し, 呼び出されているメソッドに対してリンクを張る事によってコールグラフを作成する.

コールグラフの作成と参照をカスタマイズできるようになることで, 開発者は特定のメソッドから呼び出されているメソッドのみを表示させたり, 特定のメソッドを呼び出しているメソッドの一覧を表示させたりすることができるようになり, プログラム解析の幅を広げることができるようになる.

4. 実装

本節ではまず, Mollis のアーキテクチャについて述べ, 次に具体的な解析処理の流れと提供する API の詳細について述べる.

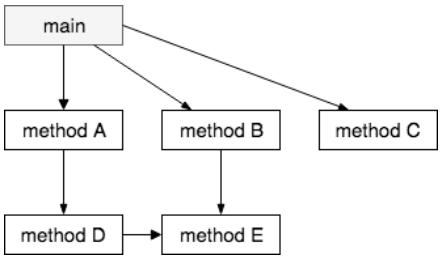

図 4 コールグラフの例

4.1 Mollis のアーキテクチャ

図 5 に示すように, Mollis は解析処理の呼び出しを行う層, デフォルトの実装が備わったインタフェースが所属する層, カスタマイズを受け付けるための実装クラスが所属する層が存在する. 以下で各層について述べるが, 機能を判別しやすくなるよう, 1 層目は呼び出し層, 2 層目はインタフェース層, 3 層目はカスタマイズ層と呼ぶ.

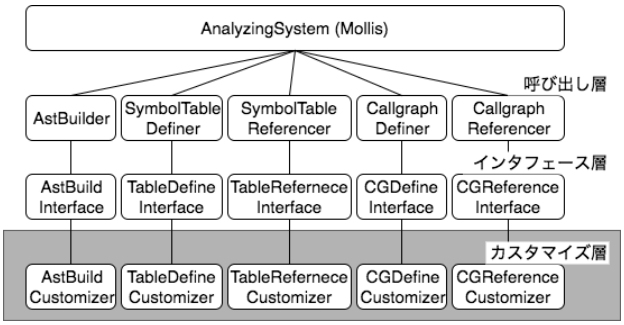

図 5 Mollis のアーキテクチャ

4.1.1 呼び出し層

前述したように，Mollis は抽象構文木の作成, 記号表の作成, 記号表の参照 (解決), コールグラフの作成, コールグラフの参照, という 5 つの段階に分割して解析処理を実行する.

4.1.2 インタフェース層

インタフェース層の各コンポーネントには, 開発者からのカスタマイズを受け付けるために, 様々なメソッドのインタフェースを用意している. Listring 1 は, AstBuildInterface の一部のメソッドを示したものである. デフォルト実装の提供のために, Java の機能の一つであるデフォルトメソッドを利用している. デフォルトメソッドを利用することで, インタフェースのメソッドに処理を定義することができる.

```
default void enterMethodName(Java8Parser.MethodNameContext ctx) {
  MethodNameNode node = new MethodNameNode(ctx.start);
  setRelation(node);
}
```

Listing 1 インタフェースの例

4.1.3 カスタマイズ層

カスタマイズ層の各コンポーネントには, それぞれのステップに対応したインタフェースの実装クラスが用意されている．Listing 2 は, Listing 1 に記述されているデフォルトの処理をカスタマイズしている例である. 開発者が Mollis をカスタマイズするときに扱うのは, カスタマイズ層に用意されている実装クラスのみである. デフォルトメソッドを開発者が実装した場合はその処理が優先されるが, 実装は強制されておらず, 実装しない場合はデフォルトの処理が実行される.

```
@Override
public void enterMethodName(Java8Parser.MethodNameContext ctx) {
    System.out.println("MethodName: " + ctx.getText());
    AstBuildInterface.super.enterMethodName(ctx);
}
```

Listing 2 カスタマイズの例

HF もうちょっと意味のあるカスタマイズにしたらどう？このカスタマイズで何が起こる？恐らく，何かが省略されるんだよね.. → 評価のとこでカスタマイズ例あるんで簡単なのでよいかと思ったんですがダメですかね...？ □

4.2 API の具体例

本節では， カスタマイズ層で開発者が使用できる API の具体的な例について示す. Mollis の処理は, 抽象構文木を作成した後の 4 つのステップは作成された抽象構文木を走査して行う為, 抽象構文木の作成時とそれ以外のステップでは解析処理が変わる. そのため, 提供する API も変化するので, 以下では作成時と作成後に分割して述べる.

4.2.1 抽象構文木の作成時

- enter, exit メソッド
  enter メソッドと exit とメソッドは, 構文規則毎に用意された処理である．enter メソッドは, 構文解析中に特定の構文規則を発見し解析を始める時に呼び出される．一方, exit メソッドは子の構文解析を全て終えた時に呼び出される. 例えば, enterMethodName メソッドは, メソッド名を表す構文規則を発見した場合に呼び出され, MethodName の子の解析を終えた時に exitMethodName が呼び出される. これらのメソッドは図 5 における AstBuildInterface 内に, デフォルト実装が記述されており, 開発者は AstBuildCustomizer 内でカスタマイズを行うことができる.
- 各種 Context クラス
  Context クラスは, 構文規則ごとに用意された解析木ノードの情報を保持する．具体的には，解析木ノードは, 子ノードのリストや, どのような字句から解析木を生成したか等の情報を保持しており, enter, exit メソッドの引数として与えられる. 例えば, Listing 2 で示した enterMethodName メソッドには，MethodNameContext が引数として与えられる. 各種 Context クラスは図 5 における, AstBuildCustomizer 内でのカスタマイズを行う時に利用することができる.
- AstBuildHelper クラス
  AstBuildHelper クラスは, 抽象構文木を作成するために必要な情報を記録する役目を担う．抽象構文木の根ノードや現在ノードの情報は, このクラスを利用して設定, 取得することができる．カスタマイズする開発者が, 抽象構文木の構造を変更する場合には, このクラスの情報と情報の扱い方を理解する必要がある.AstBuildHelper は, デフォルトの実装でも利用しているが, カスタマイズの時も開発者が自由に利用することができる. 使用する際は, AstBuildHelper も Context と同様に, AstBuildCustomizer 内で自由に扱うことができる.

4.2.2 抽象構文木の作成以降

- 各種 visit メソッド
  visit メソッドは, 抽象構文木ノードの種類ごとに用意された処理であり, 抽象構文木を走査する中で, 特定のノードを訪問した時に呼び出される．引数の型によって呼び出されるメソッドが変わるように, Visitor パターンと呼ばれるデザインパターンを利用して設計している．例えば, AdditiveExpressionNode は, 和や差を表す抽象構文木ノードであるが, このノードに対応する visit メソッドは visit(AdditiveExpressionNode node) となる. 各種 visit メソッドは AstVisitor というインタフェース内に定義されており, 図 5 における, TableDefineInterface, TableReferenceInterface, CGDefineInterface, CGReferenceInterface の 4 つのインタフェースで継承して, デフォルト実装をそれぞれ施している. 開発者は, それら 4 つのインタフェースに対応したカスタマイズ層のクラスで自由に各種 visit メソッドの処理を変更することができる.
- 各種 ASTNode クラス
  ASTNode クラスは, 抽象構文木ノードの種類ごとに用意された抽象構文木の情報を示している．その情報の例として, 親ノードや子ノードへの参照や, 木全体における深さの情報, 記号表への参照の情報等が挙げられる．ASTNode は, 各種抽象構文木ノードに共通する親クラスとなっていて, Mollis で扱う抽象構文木ノードは全て ASTNode を継承している．例えば, visit メソッドの説明の際に用いた AdditiveExpressionNode も, ASTNode クラスを継承したクラスである．各種 ASTNode は Context と同様に, visit メソッドの引数として与えられる．visitAdditiveExpressionNode の引数は, AdditiveExpressionNode である.

- Helper クラス
  Helper クラスは, Mollis が行う様々な解析ステップに存在しており, 解析処理を行うために必要な情報を記録する役目を担っている. 例えば, 記号表を作成する段階で使用する TableDefineHelper は図 5 における, TableDefineCustomizer でカスタマイズに使用することができる. また, コールグラフの作成と参照の段階で使用する CGHelper は, 図中 CGDefineCustomizer と CGReferenceCustomizer の 2 つのカスタマイズ層のクラスで使用することができる.

5. 実験および評価

本節では, 本研究で実装した Mollis の評価を行う. Mollis で評価する項目は, 柔軟性と実行速度の 2 点である. 初めに, Mollis の柔軟性が十分であるかを元に定性的に評価し, 次に実行速度を定量的に評価する.

5.1 抽象構文木のカスタマイズ

前節で述べたように, Mollis では解析木を抽象構文木へと書き換える処理をカスタマイズする事ができる. ここでは, 解析木を走査する時に指定したパッケージに属したファイルのみ抽象構文木を作成し, 不要なファイルは解析しないようにカスタマイズを行う. Listing 3 は, パッケージ名が"target"のファイルのみ抽象構文木を作成する例である. このカスタマイズを行う事で, 記号表やコールグラフを作成するために走査する抽象構文木ノードの数が減少し, 実行時間の短縮が期待できる.

```
@Override
public void exitPackageName(Java8Parser.PackageNameContext ctx){
    if (ctx.getText().equals("target")) {
        AstBuildInterface.super.exitPackageName(ctx);
    } else {
        helper.current = new EmptyNode();
    }
}
```

Listing 3 "抽象構文木のカスタマイズ"

以下に, このカスタマイズを行うために利用した Mollis の API について述べる.

- exitPackageName メソッド
  exitPackageName メソッドは, パッケージ名を表す構文を脱出する時に呼び出されるメソッドである. 今回行うカスタマイズでは, 指定されたパッケージ名のファイル以外は解析対象から外す事を目標としており, exitPackageName メソッドが呼び出されるタイミングでその処理を行う.
- PackageNameContext クラス
  PackageNameContext クラスは, パッケージ名を表す解析木の情報である. 今回のカスタマイズでは, PackageNameContext クラスが備えている getText() メソッドを利用して, 解析木が記録しているパッケージ名の情報を取得し, "target"という文字列と比較している.
- AstBuildInterface インタフェース
  AstBuildInterface は, 今回のカスタマイズ時に実装するインタフェースである. Listing 3 では, AstBuildInterface.super.exitPackageName(ctx); と記述しているが, これはインタフェースに記述されたデフォルト処理を実行するためのコードである. すなわち, 今回のカスタマイズでは if 文内の条件が真である場合, デフォルトの処理 (抽象構文木を通常通り作成する処理) を行っている.
- helper インスタンス
  helper は, AstBuildHelper クラスのインスタンスであり, 今回のカスタマイズでは現在ノードの情報を扱っている.Listing 3 では helper.current = new EmptyNode(); と記述しているが, EmptyNode は子ノード等の情報を持たない空の抽象構文木ノードを表している. すなわち, if 文内の条件が偽であるなら, 現在ノードが空の抽象構文木ノードを指すこととなり, プログラム全体の情報を参照できなくなるので, 木の構造が簡略化される.

5.2 コールグラフのカスタマイズ

Mollis が提供する API を利用することで, コールグラフの作成もカスタマイズをすることができる.Listing 4 は, メソッドの呼び出しが連鎖するソースコードである. 一般的な解析ツールでは, foo が main から呼び出されるか否かに関わらず, bar は呼び出されると判定される. しかし, main から foo が呼び出されない限り, bar は実行されることがなく, プログラマの期待と解析の結果に齟齬が生じてしまう可能性がある.

```
void main() {
}
void foo() {
  bar();
}
void bar() {
}
```

Listing 4 "メソッドの呼び出し"

そのため, プログラム実行時に呼び出されないメソッドの一覧を表示させたい場合は既存の処理をカスタマイズする必要がある. Listing 5 は, Mollis のコールグラフ作成処理に対して, プログラム実行時に呼びだされない (実行されることがない) メソッドの一覧を表示させるようなカスタマイズを行う例である.

以下に, このカスタマイズを行うために利用した Mollis

```
@Override
public void visit(MethodDeclaratorNode node) {
    if (node.text().equals("main")) {
        CGBuildInterface.super.visit(node);
    }
    return null;
}
```

Listing 5 ”コールグラフのカスタマイズ”

の API についての述べる.

- visit メソッド
  visit は, メソッドの宣言文を表す抽象構文木ノードへの到達時に呼び出されるメソッドである. 今回のカスタマイズでは, main 文のみがコールグラフにおける根ノードになるように処理を変更することを目的としており, visit(MethodDeclaratorNode node) でその処理を行う.
- MethodDeclaratorNode クラス
  MethodDeclaratorNode は, メソッドの宣言文を表す抽象構文木ノードを表すクラスである. 今回のカスタマイズでは, text() メソッドを呼び出す事によってメソッド名を取得することができるため, ”main”というメソッド名かどうかを調べている.
- CGBuildInterface インタフェース
  CGBuildInterface は, 今回のカスタマイズの時に実装するインタフェースである. 図 5 に示したように, コールグラフの作成処理をカスタマイズする場合は, CGBuildInterface を実装している CGBuildCustomizer クラスに Listing 5 のように記述する必要がある. 今回のカスタマイズでも, if 文でデフォルト処理を呼んでいるが, CGBuildInterface 内に記述されているデフォルト処理は, メソッドをコールグラフの根ノードとして登録し, そのメソッドが呼び出している別のメソッドに対してリンクを作成する処理を行っている.

5.3 柔軟性に対する考察

Mollis の柔軟性の評価として, 2 つのカスタマイズ例を示した. Listing 3 や 5 で示したように, Mollis をカスタマイズする場合には開発者が自身の要求に沿うようにオーバーライドする解析処理を判断し, 提供されている API をどのように使用するかを決めなければならない. しかし, 解析処理は細かくカスタマイズする事が可能であり, API を正しく利用すれば数行のソースコードの追加で, 要求を達成する事ができる. すなわち, Mollis の柔軟性とカスタマイズの容易性は既存のシステムと比べて高いと考えられる.

5.4 実行時間の評価

Mollis は柔軟性の向上を目的として提案, 実装したが, 実行時間が極端に増加してしまえば実用が困難となってしまう. その為, 実行時間の計測を行い, 実用に耐えうるかを評価する. Mollis の実行時間を評価するために, Mollis のカスタマイズを無効にした状態で, プログラムの静的解析の実行時間の計測を行った. 表 1 は, オープンソースソフトウェアである Apache Ant[6] のソースコードに対して, 解析を行った結果である. 解析対象は, Apache Ant の attribute ディレクトリに含まれる 6 つのファイルのセットと, helper ディレクトリの ProjectHelper2.java という単一のファイルの 2 つである. attribute ディレクトリのファイルは, AttributeNamespace.java, BaseIfAttribute.java, EnableAttribute.java, IfBlankAttribute.java, IfSetAttribute.java, IfTrueAttribute.java の 6 つである. 実験結果より, 解析対象の総コード行数が 4.66 倍になると, 実行時間は 4.70 倍になることが分かる. また, 270 行のソースコードを解析した際に, 構文解析にかかった時間を計測すると, 全体の解析時間の約 95%が構文解析に使用された時間であることがわかった.

表 1 Mollis の実行時間

解析対象	コード行数 (行)	実行時間 (ms)
attribute ディレクトリ	270	7,683
ProjectHelper2.java	1,258	36,142

5.5 実行時間に対する考察

実験結果より, 実行時間は線形に増加することが示された. すなわち, 解析対象のコード行数が増えた場合でも極端に実行時間が増加することはないと考えられる. しかし, 実行時間が線形に増加しても, 10 万行のソースコードの解析に 470 秒程度掛かってしまうことになり, これは一般的な解析ツールに比べると非常に長い解析時間である. 実験の中で, 解析に掛かった時間の内訳を割り出したところ, 構文解析に要した時間が全体の約 95%であることがわかった. Mollis で使用している構文解析器は ANTLR が自動生成したものを使用しており, 解析時間をより短縮させるためには, Mollis に最適化した構文解析器を新たに設計する必要があると考えられる.

6. まとめと今後の課題

近年のソフトウェア開発では, 大規模化や高機能化に伴い, ソフトウェアのソースコード行数も増大しており, 手作業でソースコードの評価を行うことは困難になっている. その為, ソフトウェアの安全性や開発効率の向上目的から, 静的解析ツールが用いられている. しかし, 従来の静的解析ツールは, 開発者からのカスタマイズを受け付けていない場合や, 受け付けている場合でもカスタマイズできない処理がある等, 柔軟性が高いとは言えなかった.

そこで本研究では, オープン実装に基づいた静的解析ツール，Mollis の提案と実装を行った．Mollis は, 静的解析の処理を 5 つのステップに分割し, それぞれのステップでデフォルトの処理とその処理をカスタマイズする API を開発者へと提供し, 柔軟性の向上を図った. 開発者は, 提供された API を利用することで, 様々なカスタマイズを行い, 既存の静的解析ツールでは実現できなかった要望にも対応することができる.

そして，カスタマイズの例を示すことで Mollis の柔軟性を示し，実験によって実行速度を示した．今後の課題としては, 解析速度を向上や，更なるカスタマイズを想定した API の考案が挙げられる．また, Mollis はコンソールへ警告を出力しているが, 更にユーザビリティを高める為には GUI の設計を行う必要がある.

参考文献

[1] MICHAEL. E. FAGAN, “Advances in Software Inspection,” IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL.SE-12, NO.7, pp.744-751.
[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. IEEE Software, 25(5):22-29, 2008.
[3] eclipse 公式ページ (online)
入手先 ⟨https://www.eclipse.org⟩
[4] Gregor Kiczales, John Lamping, “Open Implementation Design Guidelines,” Proceedings of the 1997 (19th) International Conference on Software engineering, 1997, pp.481-490.
[5] T. Parr. The Definitive ANTLR 4 Reference 2nd. Pragmatic Bookshelf, 2013.
[6] Apache Ant リポジトリ (online)
入手先 ⟨https://github.com/apache/ant⟩