一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

# ASTベースの静的解析ツールを対象とした多言語拡張可能なフレームワークの提案

石村　涼介† 　松下　誠†† 　肥後　芳樹††

† 大阪大学基礎工学部情報科学科 〒560-8531 大阪府豊中市待兼山町 1-3
†† 大阪大学大学院情報科学研究科 〒565–0871 大阪府吹田市山田丘 1–5
E-mail: †{r-isimur,matusita,higo}@ist.osaka-u.ac.jp

**あらまし**　抽象構文木 (AST) ベースの静的解析ツールの多くは，単一のプログラミング言語が対象であり，多言語への対応には大きなコストがかかる．一方，コンパイラの分野では，LLVM のように，言語非依存な中間表現を用いることで多言語に対応している．しかし，このようなコンパイラ基盤を，静的解析に応用することは容易ではない．本研究では，静的解析処理を言語依存部分と言語非依存部分に分離することで，多言語に拡張可能な静的解析の基盤となるフレームワークを提案する．具体的には，言語非依存なデータ構造として拡張 CST(E-CST: Extended Code Structure Tree) を導入し，フレームワークを設計した．実際に，既存の静的解析ツールである REPFINDER と PYREF に対してフレームワークを適用することで，単一言語を対象とする静的解析ツールを低コストで多言語に拡張できる可能性を示した．

**キーワード**　静的解析，抽象構文木 (AST: Abstract Syntax Tree)

## 1. ま え が き

静的解析とは，プログラムを実行せずにソースコードを解析する手法である．リンタ (Linter) はその代表例であり，主に構文規則やコーディング規約に基づく検査を行うことで，コード品質の向上や，コードレビューのコスト削減に寄与している．例えば，Python を対象とした flake8(注1) や mypy(注2) などのリンタが開発現場において広く利用されている．

一方，近年の研究では，単純な構文規則やコーディング規約の検査にとどまらず，コード要素間の関係やリビジョン間の変化といった，より高度な構造情報を用いる静的解析ツールが提案されている．例えば，PYREF [1] は，Python プロジェクトにおいて，リビジョン間のリファクタリングを検出する静的解析ツールである．REPFINDER [2] は，Java プロジェクトにおいて，ライブラリ更新時に欠落した API の代替となる API を探索する静的解析ツールである．

静的解析では，解析の目的に応じて用いられるデータ構造が異なる．例えば，抽象構文木 (AST: Abstract Syntax Tree) は，プログラムの構文構造を木構造として表現したデータ構造である．AST は構文規則やコーディング規約の検査，型解析など，多くの静的解析で用いられる．一方，制御フローグラフ (CFG: Control Flow Graph) は，文や基本ブロック間の実行順序をグラフとして表現したデータ構造である．CFG はデータフロー解析や抽象解釈など，制御構造を考慮した解析に用いられる．

このように，多種多様な静的解析ツールが提案・利用されているが，多くの静的解析ツールは単一のプログラミング言語を対象としており，多言語への拡張には大きなコストを要するという課題がある．これは，解析処理の大半が対象言語に依存して実装されており，言語間で再利用可能な部分が限定的であるためである．したがって，新たに別のプログラミング言語に対応する際には，解析器の新規実装が必要となり，大きなコストを要する．

一方，コンパイラの分野では，多くのプログラミング言語に対応可能なコンパイラ基盤として LLVM [3] が知られている．LLVM では，コンパイル対象となるソースコードを，プログラミング言語に依存しない中間表現である LLVM IR へ変換する．その後の最適化や機械語生成などの処理は LLVM IR 上で行われるため，言語間で共通の処理を再利用できる．また，新たなプログラミング言語に対応する際には，当該言語から LLVM IR への変換部分のみを新規実装すればよいため，低コストである．

LLVM のようなコンパイラ基盤と同様に，静的解析においても基盤を構築することで，多言語への拡張を低コストで実現できると考えられる．LiSA [4] は，CFG に基づく静的解析基盤であり，現時点では Java，Python，Go を対象言語としている．LiSA では，解析対象となるソースコードを，プログラミング言語に依存しない中間表現である LiSA CFG へ変換する．以降の解析処理は LiSA CFG 上で行われるため，LLVM と同様に，言語間で共通の処理を再利用できる．

このように，CFG に基づく静的解析ツールを対象とした基盤は存在するが，AST に基づく静的解析ツールを対象とした基

（注1）：https://flake8.pycqa.org
（注2）：https://mypy-lang.org

盤は発展途上である．そこで本研究では，AST に基づく静的解析基盤として，拡張 CST(E-CST: Extended Code Structure Tree) を提案する．E-CST は，RefDiff 2.0 [5] で提案された CST(Code Structure Tree) を拡張した言語非依存なデータ構造であり，RefDiff 2.0 以外の静的解析ツールにも適用可能である．

実際に，既存の静的解析ツールである REPFINDER と PYREF に対して E-CST を適用することで，単一言語を対象とする静的解析ツールを低コストで多言語拡張できる可能性を示す．

## 2. 準　　備

本節では，研究の背景として，静的解析，RefDiff，CST について述べる．

### 2.1 静 的 解 析

静的解析とは，プログラムを実行せずにソースコードを解析する手法である．リンタ (Linter) はその代表例であり，主に構文規則やコーディング規約に基づく検査を行うことで，コード品質の向上や，コードレビューのコスト削減に寄与している．例えば，Error Prone [6] は Google が開発した Java 言語向けのリンタであり，コンパイル時に AST を解析することで，実行時にエラーを引き起こす可能性のあるバグパターンを検出する．

静的解析を実現するためには，解析対象となるソースコードを何らかのデータ構造に変換する必要がある．代表的なデータ構造として，AST や CFG が挙げられる．AST は，プログラムの構文構造を木構造で表現したもの [7] であり，構文規則の検査やリファクタリング検出などに広く用いられている．一方，CFG は基本ブロックをノードとし，制御の流れをエッジとする有向グラフ [8] であり，データフロー解析や抽象解釈などに用いられる．

多くの静的解析ツールは，特定のプログラミング言語を対象として実装されている．これは，データ構造の構築や解析処理が対象言語の構文や仕様に強く依存しており，複数言語を同一の枠組みで扱うことが容易ではないためである．その結果，既存の静的解析ツールを多言語に拡張する場合，大きなコストが発生するという課題が指摘されている [9].

このような背景から，言語に依存しない共通のデータ構造を用いることで，静的解析ツールの多言語対応を低コストで実現する試み [4] [9] [10] が注目されている．

### 2.2 RefDiff

RefDiff [11] とは，Java プロジェクトにおいて，異なるリビジョン間のソースコードを比較し，リファクタリングを自動的に検出する静的解析ツールである．例えば，メソッド M がクラス X からクラス Y へ移動された場合，RefDiff はその変更を Move Method として検出する．

このようなリファクタリング検出においては，条件文や式の詳細な構文構造よりも，コード要素の包含関係や名前，位置関係といった構造的情報が重要となる．RefDiff では，Java ソースコードから AST を構築し，構築した AST からクラス，メソッド，フィールドなどの情報を抽出することで解析を行う．

当初，RefDiff は Java 言語のみを対象としていたが，RefDiff 2.0 では多言語に拡張され，Java，JavaScript，C の 3 言語に対応可能となった．RefDiff 2.0 では，多言語対応を実現するために，CST と呼ばれる言語非依存なデータ構造が導入された．

```java
package com.ex;
public class Main {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        int r = c.plus(2, 5);
        System.out.println(r);
    }
}
```

図 1　Java コードの例 (com/ex/Main.java)

```java
package com.ex;
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }

    /**
     * @deprecated Use add(int, int) instead.
     */
    @Deprecated
    public int plus(int x, int y) {
        return x + y;
    }
}
```

図 2　Java コードの例 (com/ex/Calculator.java)

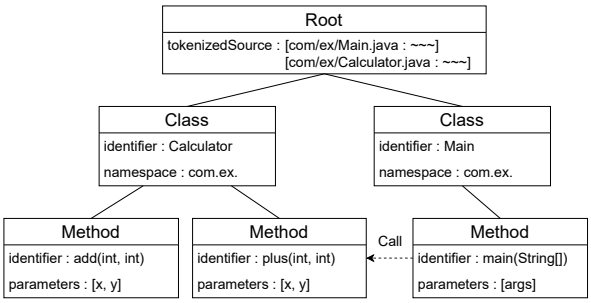


図 3　CST の 例

### 2.3 CST

CST とは，RefDiff 2.0 において導入された言語非依存なデータ構造であり，Java，JavaScript，C のいずれかの AST をもとに構築される．CST は AST と同様に木構造でプログラムを表現するが，すべての要素を表現するのではなく，クラスやメソッド，関数といった要素のみに着目する．一方，条件文や式などの詳細な要素は抽象化され，リファクタリング検出に必要な情報に限定した表現となっている．

図 1，図 2 のソースコードから得られる CST の例を図 3 に示す．CST は 1 リビジョンにつき 1 つ生成されるため，図 3 では同一リビジョンに属する 2 つのソースコードから 1 つの CST が生成されている．CST のルートノードは，各ソースコードをトークン単位に区切った状態で保持し，類似度の計算に使用する．また，ルートノード以外のノードは，クラスやメソッド

表 1　CST 上で表現される AST ノード [5]

| 言語 | AST ノード |
|---|---|
| Java | class, enum, interface, and method |
| C | file and function |
| JavaScript | file, class, and function |

表 2　E-CST が保持する情報

| 情報 | 例 | 備考 |
|---|---|---|
| tokenizedSource | [”private”, ”void”, ...] | ルートノードのみ |
| identifier | ”plus(int, int)” | |
| namespace | ”com.ex.” | クラス，ファイルのみ |
| parameterNames | [”x”, ”y”] | 関数，メソッドのみ |
| parameterTypes | [”int”, ”int”] | 関数，メソッドのみ |
| returnType | ”int” | 関数，メソッドのみ |
| deprecatedMessage | ”Use hoge() instead.” | |

を表現しており，識別子 (例:plus(int, int)) を持つ．クラスを表現するノードには名前空間 (例:com.ex.) が付与され，メソッドを表現するノードには引数名リスト (例:[x, y]) が付与される．このような設計により，クラスやメソッドの移動や名称変更といったリファクタリングを効率よく検出できる．

また，ノード間には，継承関係や呼び出し関係などの関係が明示的に付与されており，クラス階層や依存関係に基づく解析を効率的に実施できる．例えば，図 1 における main メソッドは図 2 における plus メソッドを呼び出しているため，図 3 では main メソッドを表現するノードから plus メソッドを表現するノードへ有向辺が付与されている．

CST のノードは，各プログラミング言語の AST ノードから生成されるが，その表現は言語非依存となるよう設計されている．CST 上で表現される AST ノードの一覧を表 1 に示す．例えば，Java ではクラスやメソッドが CST のノードとして表現される一方，JavaScript や C においても，それぞれの言語における関数やファイルが共通の概念として CST 上に表現される．すなわち，CST は言語間の差異を吸収する中間表現であるといえる．

なお，リビジョン間で追加・削除・変更されたファイルのみを対象として CST が構築される．未変更のファイルはリファクタリング検出において重要な役割を果たさないためである．

## 3. 提案フレームワーク

本フレームワークの目的は，静的解析処理を言語間で共通化することである．そのために，言語非依存なデータ構造として，CST を多様な静的解析でも利用できるように拡張した CST（以降 E-CST）を導入する．

### 3.1 E-CST

E-CST とは，RefDiff 2.0 において導入された CST を，リファクタリング検出に限らず，より一般的な AST ベースの静的解析にも適用可能とした言語非依存なデータ構造である．CST が持つ言語非依存性を維持しつつ，より多様な静的解析に必要となる情報および操作を追加している．

表 3　E-CST が提供する操作

| 操作 | 備考 |
|---|---|
| 全ノードの走査 | 深さ優先探索 (pre-order のみ) |
| 関係グラフを辿る | 呼び出し関係または継承関係 |
| ノードをソースコードに復元 | |
| ソースコード全文の復元 | |
| Visitor を accept する | Visitor パターン用 |

#### 3.1.1 保持する情報

E-CST が保持する情報を表 2 に示す．E-CST では，CST に対して以下の情報を追加した．

- 引数型リスト (parameterTypes)
- 返り値の型 (returnType)
- 非推奨メッセージ (deprecatedMessage)

CST では引数名リスト (parameterNames) のみ保持していたが，E-CST では引数および返り値の型情報を保持することで，メソッド探索や API 間の比較において型情報を用いることができる．また，非推奨メッセージを保持することで，非推奨 API の代替候補探索や API 移行支援などへの応用が期待できる．

なお，表 2 に示した情報は，文字列型として保持する．identifier，namespace，returnType，deprecatedMessage は文字列型として保持する．parameterNames および parameterTypes は文字列型の順序付きリストとして保持し，引数宣言順に格納する．tokenizedSource はトークンを表す文字列型の順序付きリストとして保持する．

#### 3.1.2 提供する操作

E-CST が提供する操作を表 3 に示す．E-CST では，CST に対して以下の操作を追加した．

- ソースコード全文の復元
- Visitor パターン用の accept() メソッド

まず，ソースコード復元に関する差異について述べる．CST では，単一のノードをソースコードに復元することは可能であったが，ソースコード全文を復元する操作は提供されていなかった．そこで E-CST では，ルートノードを入力とし，各ファイルパスに対応するソースコード文字列の集合を出力する復元操作を定義した．本操作は，ルートノードが保持する tokenizedSource を用いて，各ファイルのソースコードを再構成する．これにより，E-CST 上では保持していない詳細な構文情報を，必要に応じて元のソースコードから参照でき，構造情報とテキスト情報を組み合わせた解析が可能となる．

次に，走査機構の差異について述べる．CST では，ルートノードを起点とする深さ優先探索 (pre-order) を実行できる．しかしこの機構は，単一のコールバック関数を適用する形式の走査であり，ノード型ごとの処理分岐や post-order の深さ優先探索などは定義されていない．そこで E-CST では，E-CST Visitor を引数として受け取り，ノード到達時 (pre-order) および子ノード走査後 (post-order) に E-CST Visitor の対応メソッドを呼び出す accept 操作を定義した．本操作は値を返さず，解析結果は E-CST Visitor の内部状態として保持される．これにより，E-CST を変更することなく新たな解析処理を追加可能と

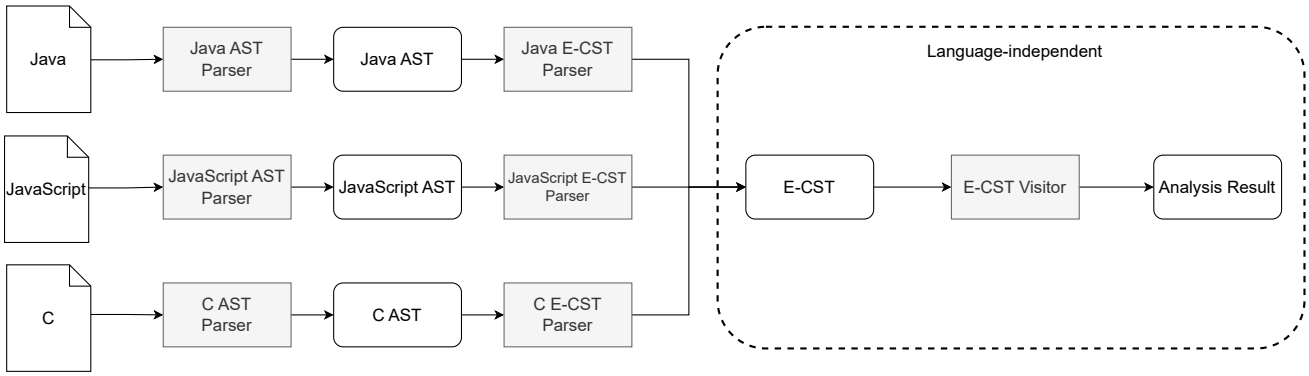


図 4　提案フレームワーク

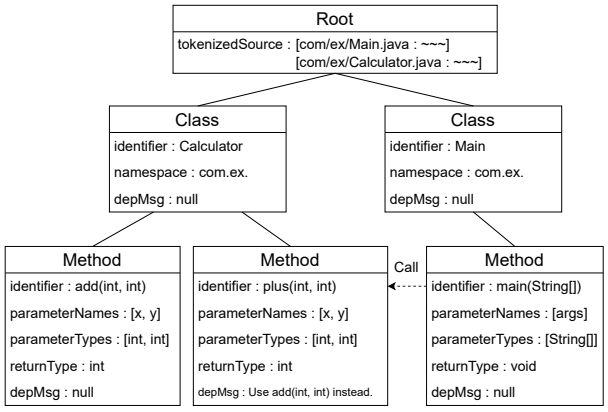


図 5　E-CST の例

なり，汎用的な AST ベースの静的解析基盤へと拡張されていることが分かる．

#### 3.1.3　E-CST の例

図 1，図 2 のソースコードから得られる E-CST の例を図 5 に示す．ここで，E-CST の木構造自体は，図 3 の CST と同様である．すなわち，E-CST は CST と同様に，1 リビジョンにつき 1 つ生成される．また，CST と同様に，main メソッドを表現するノードから plus メソッドを表現するノードへ，呼び出し関係を表す有向辺が付与されている．

一方，図 5 においてメソッドを表現するノードに注目すると，本フレームワークにおいて新たに追加された，引数型リスト，返り値の型，非推奨メッセージが保持されている．また，クラスを表現するノードに注目すると，本フレームワークにおいて新たに追加された非推奨メッセージが保持されている．

### 3.2　処理の流れ

本フレームワークは 3 段階の処理に分けられる（図 4）．

（1）　**AST の構築**

解析対象となるソースコードを AST に変換する．ここでは既存の AST パーサを用いる．例えば Java 言語の場合は，JDT(注3) の AST パーサを用いる．

（2）　**E-CST の構築**

AST をもとに E-CST を構築する．例えば Java の AST をもとに E-CST を構築する場合，Java 専用の E-CST パーサを用いる．すなわち，E-CST パーサは解析対象となる言語ごとに開発する必要がある．

（3）　**静的解析の実行**

E-CST 上で静的解析を実行する．E-CST は言語非依存なデータ構造であるから，解析対象の言語にかかわらず，解析器を再利用できる．解析器としては，Visitor パターンに基づく E-CST Visitor を想定している．これにより，E-CST 側を変更することなく解析器を開発できる．

## 4.　提案フレームワークの適用

本節では，既存の静的解析ツールである REPFINDER と PYREF に対し，提案フレームワークの適用例を示す．

### 4.1　適 用 方 法

提案フレームワークの適用は，以下の 3 段階の手順により行う．

（1）　**E-CST への変換**

解析対象となるソースコードを E-CST へ変換する．対象言語に対応した E-CST パーサが既に存在する場合はそれを利用し，存在しない場合には，当該言語の AST を入力として E-CST を構築する E-CST パーサを新たに実装する．

（2）　**静的解析の実行**

構築した E-CST に対して静的解析を実行する．E-CST は Visitor パターンを採用しているため，解析内容に応じた E-CST Visitor を実装することで，E-CST の構造を変更することなく解析処理を追加できる．この解析器は言語非依存であり，同一の Visitor を複数のプログラミング言語に対して再利用可能である．

（3）　**解析結果の収集・利用**

解析結果を収集・利用する．E-CST 上で得られた解析結果は，リファクタリング検出や非推奨 API の検出，コード要素間の関係解析など，さまざまな静的解析に応用できる．

### 4.2　REPFINDER への適用

REPFINDER は Java プロジェクトにおいて，ライブラリ更新時に欠落した API の代替となる API を探索する静的解析ツールである．ライブラリ更新時，一部の API は削除されたり非

(注3)：https://www.eclipse.org/jdt/

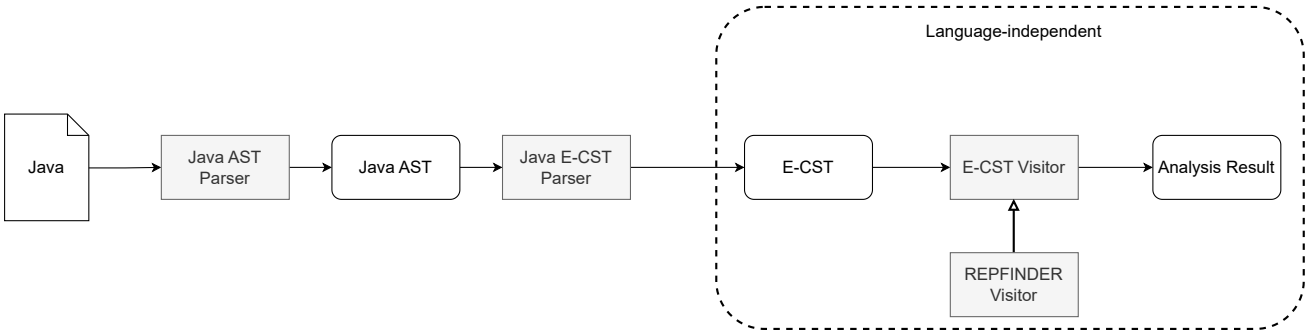


図 6　提案フレームワークの適用例 (REPFINDER)

推奨 (deprecated) とされたりするため，欠落することがある．その結果，ライブラリ利用者は欠落 API の代替 API を手動で探す必要があり，これは大きなコストのかかるソフトウェア保守作業となる．REPFINDER では代替 API を自動的に発見するために，以下の 3 段階の探索を行う．

（1） JavaDoc 非推奨メッセージをもとに探索
（2） 自ライブラリを探索
（3） 外部ライブラリを探索

3 段階の探索によって，既存手法よりも高い再現率で代替 API を提示できることが報告されている．

REPFINDER は，API 間の構造的関係や継承関係を解析するため，Java の AST に基づいた静的解析を行う．具体的には，クラス階層の探索やメソッドシグネチャの比較を通じて，欠落 API と類似した API を代替 API の候補として抽出する．すなわち，REPFINDER は Java の AST に強く依存している．

その結果，REPFINDER の探索アルゴリズム自体は他のプログラミング言語にも適用可能な性質を持つ一方で，実装上は Java 専用のツールとなっている．

REPFINDER へのフレームワークの適用例を図 6 に示す．具体的な適用手順は以下の 3 段階である．

（1） **E-CST への変換**

Java ソースコードを JDT の AST パーサにより AST へ変換し，その AST を入力として E-CST を構築する．クラス，メソッド，継承関係，呼び出し関係，および JavaDoc 非推奨メッセージを E-CST 上のノードおよび関係として表現する．

（2） **静的解析の実行**

REPFINDER の探索処理を，E-CST Visitor を用いて再実装する．E-CST Visitor を継承した REPFINDER Visitor を実装し，継承関係やシグネチャ類似性の解析を，Visitor パターンによる E-CST 走査として実現する．

（3） **解析結果の収集・利用**

欠落 API に対する代替 API 候補を抽出する．

このように，提案フレームワークを適用することで，REPFINDER の探索処理は Java の AST から独立し，E-CST に基づく言語非依存な解析処理として再構成されると考えられる．同様の E-CST パーサを他言語向けに実装することで，REPFINDER の解析器を再利用し，低コストで他のプログラミング言語へ拡張できると考えられる．

### 4.3　PYREF への適用

PYREF は，Python プロジェクトを対象として，リファクタリングを自動的に検出する静的解析ツールである．

PYREF is, RefactoringMiner [12] から着想を得た手法を Python 向けに適用し，Python の AST を用いてコード要素をモデル化することで，メソッド単位のリファクタリング検出を実現している．具体的には，RENAME METHOD，ADD PARAMETER，EXTRACT METHOD，MOVE METHOD などを検出する．

PYREF は，リビジョン間で変更されたファイルのみを対象とし，Python の AST からモジュール，クラス，メソッド，文といったコード要素を抽出する．抽出した要素をノードとして表現し，異なるリビジョン間のノード同士を対応付けることで，どのノードがどのノードに変化したか特定し，リファクタリング候補を検出する．この解析処理は，Python の AST に強く依存している．

その結果，PYREF の探索アルゴリズム自体は他のプログラミング言語にも適用可能な性質を持つ一方で，実装上は Python 専用のツールとなっている．

PYREF へのフレームワークの適用例を図 7 に示す．具体的な適用手順は以下の 3 段階である．

（1） **E-CST への変換**

Python ソースコードを，ast モジュールを用いて AST へ変換し，その AST を入力として E-CST を構築する．モジュール，クラス，メソッドといった PYREF の解析に必要なコード要素を，E-CST 上のノードとして表現する．

（2） **静的解析の実行**

PYREF が行うリファクタリング検出処理を，E-CST 上の解析として再実装する．E-CST Visitor を継承した PYREF Visitor を実装し，リファクタリング検出処理を，Visitor パターンによる E-CST 走査として実現する．

（3） **解析結果の収集・利用**

メソッド単位のリファクタリング操作を抽出する．

このように，提案フレームワークを適用することで，PYREF のリファクタリング検出処理は Python の AST から切り離され，E-CST に基づく言語非依存な解析処理として再構成されると考えられる．同様の E-CST パーサを他言語向けに実装することで，PYREF の解析器を再利用し，低コストで他のプログラミング言語へ拡張できると考えられる．

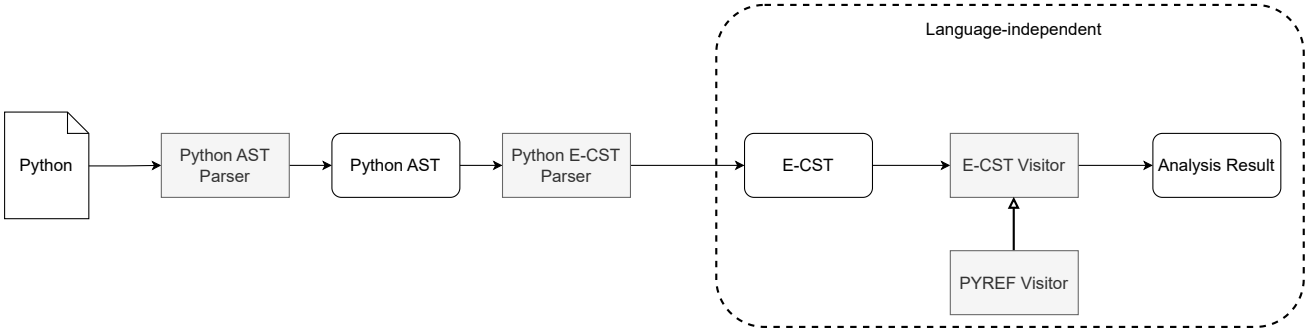


図 7　提案フレームワークの適用例 (PYREF)

## 5. 考　　察

REPFINDER および PYREF への適用を通じて，本フレームワークは，異なるプログラミング言語を対象とし，かつ異なる解析目的を持つ AST ベースの静的解析ツールに対して，共通の解析基盤を提供可能であることを示した．この結果は，本フレームワークが高い汎用性を有することを示唆している．

従来，これらの静的解析ツールはそれぞれ対象言語の AST に強く依存して実装されており，多言語への拡張は困難であった．しかし，本フレームワークを用いることで，解析処理を言語非依存な部分として切り出すことができ，また前処理となる言語依存部分を E-CST パーサに分離できる可能性が示された．

一方で，本フレームワークは AST ベースの静的解析を対象としており，すべての解析を E-CST 上で完全に表現できるわけではない．例えば，式レベルの詳細な構文解析や制御フローを考慮する解析では，E-CST のみでは情報が不足する場合がある．このような制約を踏まえると，本研究のアプローチは，CFG ベースの解析基盤である LiSA と相補的な関係であると考えられる．

また，GAST [10] のような汎用的な言語非依存 AST と比較すると，E-CST は表現する要素をクラス，メソッドなどに限定している．そのため，本フレームワークの適用対象は，AST ベースの静的解析の中でも，コード要素の包含関係や名前，位置関係といった構造的情報に注目するものに絞られる．この設計により，解析基盤としての実装負荷や解析処理の複雑さを抑えつつ，多言語拡張性を確保できる点が E-CST の特徴である．

## 6. あ と が き

本研究では，AST ベースの静的解析ツールを対象とした，多言語拡張可能なフレームワークとして E-CST を提案した．E-CST は，CST が持つ言語非依存性を維持しつつ，型情報や非推奨メッセージといった静的解析に有用な情報を拡張したデータ構造である．

また，既存の静的解析ツールである REPFINDER および PYREF に適用することで，解析処理を対象言語の AST から切り離し，言語非依存に再構成できる可能性を示した．これにより，単一言語向けに開発された AST ベースの静的解析ツールを，低コストで多言語に拡張できることが期待される．

しかし，現時点では E-CST パーサは未整備であり，特に Python 用パーサは未実装である．Java，JavaScript，C についても，RefDiff 2.0 で用いられている CST パーサは存在するものの，E-CST パーサは未実装である．今後は，E-CST パーサを実装し，実際に REPFINDER や PYREF を多言語に拡張することで提案フレームワークの有効性を示す必要がある．

**謝辞**　本研究は JSPS 科研費 25K03102, 24H00692, 23K24823 の助成を受けたものです．

## 文　　献

[1] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, “Pyref: Refactoring detection in python projects,” IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp.136–141, 2021.

[2] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, “Repfinder: Finding replacements for missing apis in library update,” Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.266–278, 2021.

[3] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO), pp.75–85, 2004.

[4] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, “Static analysis for dummies: Experiencing lisa,” Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP), pp.1–6, 2021.

[5] D. Silva, J.P. daSilva, G. Santos, R. Terra, and M.T. Valente, “Refdiff 2.0: A multi-language refactoring detection tool,” IEEE Transactions on Software Engineering, vol.47, no.12, pp.2786–2802, 2020.

[6] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” Communications of the ACM, vol.61, no.4, pp.58–66, 2018.

[7] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd edition, Addison-Wesley, 2007.

[8] F.E. Allen, “Control flow analysis,” ACM SIGPLAN Notices, vol.5, no.7, pp.1–19, 1970.

[9] 坂本一憲，大橋　昭，太田大地，鷲崎弘宜，深澤良彰，“Unicoen: 複数プログラミング言語対応のソースコード処理フレームワーク，” 情報処理学会論文誌，vol.54，no.2，pp.945–960，2013.

[10] Leiton-Jimenez and et al., “Gast: A generic ast representation for language-independent source code analysis,” ENFOQUE UTE, vol.14, no.4, pp.9–18, 2023.

[11] D. Silva and M.T. Valente, “Refdiff: Detecting refactorings in version histories,” IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp.269–278, 2017.

[12] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” IEEE Transactions on Software Engineering, vol.48, no.3, pp.930–950, 2022.