論　文

多言語対応メトリックス計測プラグイン開発基盤MASUの開発

三宅　達也[†]　肥後　芳樹[†]　楠本　真二[†]　井上　克郎[†]

MASU: A Metrics Measurement Framework for Multiple Programing Languages

Tatsuya MIYAKE[†], Yoshiki HIGO[†], Shinji KUSUMOTO[†], and Katsuro INOUE[†]

あらまし　筆者らは複数の言語に対して適用可能なメトリックス計測プラグインの開発基盤 MASU を開発している．MASU はソースコードを解析し，メトリックス計測に必要な情報をユーザに提供する．また，計測するメトリックスに応じたソースコード解析や新規メトリックスへの対応を容易にするため，MASU はソースコードから抽出する情報を柔軟に変更・追加できるように設計されている．ユーザは MASU が提供する情報を利用することにより，必要最低限のメトリックス計測ロジックを記述するだけでメトリックスを計測できる．本論文では MASU の詳細について述べる．

キーワード　ソフトウェアメトリックス，メトリックス計測ツール，ソースコード解析

1. ま え が き

ソフトウェアメトリックスとは，ソフトウェアの品質評価に用いられる尺度であり [25]，ソースコードなどのソフトウェアプロダクトから計測される．ソフトウェアメトリックスには単純にソースコードの行数を表す LOC から，オブジェクト指向言語におけるクラスやメソッド，フィールド間の関連性を評価する CK メトリックス [6]，関数やメソッド内の処理の複雑さを表すサイクロマチック数 [23] などの様々な種類が存在する．これらのメトリックスはソフトウェアの概念的な要素に対して定義されているため，プログラミング言語間の記述様式の差異にとらわれることなく，同じ概念を共有する言語に共通して適用することができる．

しかし，メトリックスを計測するために，ソースコードを解析しソフトウェアの概念的な要素の情報を取得するには大きなコストを必要とする．これまでに種々のコンパイラコンパイラ [1], [17] が開発されているが，それらが支援しているのは抽象構文木構築などの構文解析までであり，変数の参照・代入や関数の呼出しなど，どのようにプログラム内の要素が関連しているかを調査するためには，開発者自らが意味解析を行う処理を実装しなければならない．

また，メトリックスを計測するツールの多くは単一言語を対象としているため [3], [7], [16], [18], [28]，様々な言語のソースコードのメトリックスを計測するには，それぞれの言語に対応した計測ツールを個別に用意する必要がある．しかし，ツールごとに対応しているメトリックスの種類は異なる．また，同一のメトリックスに対応している計測ツールであっても定義のあいまいな部分の解釈はツールごとに異なる．このため，複数の言語から統一的にメトリックス値を計測することは難しい．

これらのことを踏まえると，以下の特徴をもつメトリックス計測ツールが必要であるといえる．

- 複数のプログラミング言語のソースコードに対して適用可能である．
- 多言語の解析から得た結果を統一的に扱える．
- ユーザは，必要最低限のメトリックス計測ロジックを記述するだけで，新たなメトリックスを計測できる．

本論文では，これらの要求を満たすべく我々が開発しているメトリックス計測プラグイン開発基盤 MASU [22] を紹介する．

2. 背　　景

2.1 ソフトウェアメトリックス

ソフトウェアメトリックスとは，ソフトウェアの品質評価や工数・保守コスト予測などに用いられる尺

[†] 大阪大学大学院情報科学研究科，豊中市
Graduate School of Information and Science Technology, Osaka University, Toyonaka-shi, 560–8531 Japan

 ⓒ（社）電子情報通信学会 2009

度である [25]．代表的なソフトウェアメトリックスには，オブジェクト指向言語におけるクラスやメソッド，フィールド間の関連性からソフトウェアの複雑度を評価する CK メトリックス [6]，関数やメソッド内の処理の複雑さを表すサイクロマチック数 [23] などが存在する．

多くのソフトウェアメトリックスは，CK メトリックスやサイクロマチック数のようにファイルやクラス，メソッド，関数，制御フローといったソフトウェアの概念的な要素に対して定義されるものであるため，プログラミング言語間の記述様式の差異にとらわれることなく，同じ概念を共有する言語に共通して適用できる．

ソフトウェアメトリックスの多くは厳密な定義がされていない．例えば，CK メトリックスの一つである WMC はクラス当りの重み付きメソッド数を表すが，メソッドの重みに関する厳密な定義は存在しない．このような，ソフトウェアメトリックスの定義のあいまいさはメトリックス値の計測結果に大きな影響を及ぼす．

また，既存のソフトウェアメトリックスが必ずしも最善であるとは限らない．このため，既存のメトリックスの改善案が提案されたり，同じ目的で使用される新規メトリックスが提案されることもある．例えば，CK メトリックスの一つである LCOM は Chidamer と Kemerer による定義では，メソッドの凝集の欠如を正確に表せない場合があることが知られており，異なる定義をもつ LCOM もいくつか提案されている [13], [14].

2. 2　ソフトウェアメトリックス計測

ソースコードを対象としたメトリックス計測は基本的に次の 2 ステップに分けることができる．

［ソースコード解析］　メトリックス値の計測に必要な情報をソースコードから抽出．

［メトリックス値計測］　ソースコードから抽出した情報を利用してメトリックス値を計測．

2. 1 で述べたように，ソフトウェアメトリックスはプログラミング言語間の記述様式の差異にとらわれることなく，同じ概念を共有する言語に共通して適用できる．このため，メトリックス値計測のロジックは言語が異なっても共通して利用することができる．一方，異なる記述様式に対応するためには，各記述様式ごとにソースコード解析器を用意しなければならない．

既存の研究や開発において様々なメトリックス計測ツールが開発されている [3], [7], [16], [18], [28]．しかし，既存のツールを用いて，異なる言語で記述された複数のソフトウェアから統一的にメトリックス値を計測することは難しい．これは次の問題に起因する．

(a)　ソースコード解析器は言語ごとに用意する必要があるが，ソースコード解析器の実装は高いコストを必要とするため，既存のメトリックス計測ツールの多くは単一の言語を対象としている．

(b)　問題 (a) のため，異なる言語で記述された複数のソフトウェアのメトリックス計測を行うには，それぞれの言語に対応した複数のメトリックス計測ツールを用意する必要がある．しかし，ソフトウェアメトリックスの定義はあいまいなものが多いため，既存のメトリックス計測ツールは，同一のメトリックスであってもツールごとに計測結果が大きく異なる [19].

また，メトリックスごとに，計測に必要とするソースコードの情報は異なるため，新規メトリックスの計測において既存のツールのソースコード解析を再利用することは難しい．

このように，メトリックス計測において発生する問題は，ソースコード解析に対する支援不足が大きな原因の一つとなっている．

2. 3　ソースコード解析

ソースコード解析とは，ソースコードを解析器と呼ばれるプログラムを用いて，自動で必要な情報を抽出する技術である．抽出された情報は，ソフトウェアメトリックスやその組織独自の方法を用いて調査され，ソフトウェアの品質を評価するためなどに用いられる．

しかし，ソースコードを解析し，その情報からメトリックスなどを算出するには，例えば，関数間の呼出し関係などの，意味解析以上の深い解析をしなければ得ることのできない情報を必要とするため，解析器自体を作成することに高いコストが必要である．既に，JavaCC [17] や ANTLR [1] などに代表される多くのコンパイラコンパイラが開発されているが，それらが支援するのは構文解析までであり，それ以上の深い解析を行う場合は，多大な時間と労力を必要とする．

一方，ソフトウェアのソースコードが調査される機会は増加しており，ソースコード解析技術の重要性は増している．例えば，ソフトウェア開発企業では，QA (Quality Assurance) のために，開発したソフトウェアをソフトウェア工学的な手法を用いて調査することが増えてきた．この調査の一環として，ソースコード解析を行い，その品質や今後の保守コストなどを予測することが多い．しかし，近年顕著に見られる開発期

間の短縮化や，対象ソフトウェアによって適用する解析手法が異なることから，満足に QA が行われているとはいえない．

このような問題は，深い解析を行う解析器を実装するコストが高いことや，そのような解析器を実装できる開発者を確保することが困難であることに起因している [12]．このため，ソフトウェアの品質を評価する技術者は，品質評価を行うためのアルゴリズムの実装ではなく，そのアルゴリズムで用いる情報を抽出する解析器の実装に悩まされたり，若しくは，解析器を必要とするソフトウェアの品質評価自体を断念したりしている．

3. 要求されるメトリックス計測ツール

本章ではソフトウェアメトリックスの特徴や既存のメトリックス計測ツールの問題を考慮した上で，メトリックス計測ツールに要求されるであろう機能と，その機能を実装するための設計・実装方針について述べる．

3.1 機能要求

メトリックス計測ツールに要求される機能として次のものが考えられる．

[複数言語への対応]　複数言語に対応したツールの供給不足を解消し，ソフトウェアメトリックスを幅広く活用するために，複数の言語に適用可能なメトリックス計測ツールが必要である．

[統一的なメトリックス計測]　異なる言語で記述された複数のソフトウェアから同一のメトリックス計測ロジックでメトリックスを計測するために，異なる言語で記述された複数のソフトウェアの解析結果を統一的に扱えるメトリックス計測ツールが必要である．

[ユーザがメトリックス計測部の定義を自由に変更可能]　将来提案されるであろうソフトウェアメトリックスの計測に対応するために，計測するメトリックスをユーザが自由に追加し，その計測ロジックを自由に定義可能な環境が必要である．ただし，ユーザ参加型のシステムを構築する際には，システムの安全性を確保するために，ユーザに許可する権限を明確にし，ユーザの行う処理を管理する必要がある．

3.2 設計・実装方針

メトリックス計測ツールに要求される機能を実現するために，次に述べる設計・実装方針が必要であると筆者は考える．

[メトリックス計測部とソースコード解析部の分離]
メトリックス計測部とソースコード解析部を分離しておくことにより，異なる言語で記述されたソフトウェアのメトリックス計測の際にも，メトリックス計測部は再利用することができる．これにより，複数言語への対応に必要とするコストの削減が期待できる．また，メトリックス計測部を再利用することは，言語が異なっても同一のメトリックス計測ロジックを使用することを意味する．つまり，統一的なメトリックス計測を実現できる．

[言語間の差異の吸収]　メトリックス計測ツールの実装には，メトリックス計測部よりもソースコード解析部の実装の方が高いコストを必要とする．このため，複数言語への対応に必要とするコストを削減するには，ソースコード解析の初期段階に言語間の差異を吸収し，ソースコード解析部も可能な限り再利用することが望ましい．また，メトリックス計測部を再利用ためには，言語非依存な共通データを構築しなければならない．

[メトリックスとメトリックス計測部の 1 対 1 対応]
一つのソフトウェアメトリックスにつき，一つのメトリックス計測部が用意されることが望ましい．このような設計を適用することにより，計測対象メトリックスを追加するする際にも，新たなメトリックス計測部を追加するだけでよく，既存のメトリックス計測部を変更する必要はない．また，改善したいメトリックスに対応したメトリックス計測部を継承することにより，改善したメトリックスの計測を低コストで実現することができる．

つまり，計測対象メトリックスとメトリックス計測部を 1 対 1 対応にすることにより，ユーザが計測対象メトリックスを自由に追加したり，計測ロジックの定義を自由に変更したりすることが可能となる．

[解析対象要素を変更可能なソースコード解析器]　あるメトリックスの計測に必要な情報が，必ずしも他のメトリックスの計測に必要であるとは限らない．必要としない情報の解析は時間やメモリ空間の無駄となる．また，新規メトリックスの計測を行いたいが，必要なソースコードの情報が不足しているという状況も考えられる．

このため，ソースコード解析器には拡張・変更容易性の高い設計を適用し，解析対象要素を柔軟に追加・変更できるようにすることが望ましい．

4. MASU の仕様

3.2 で述べた方針に基づき，複数言語の解析結果から統一的にメトリックス計測を行えるプラグインの開発基盤 MASU (Metrics Assessment plugin platform for Software Unit) を開発した [22].

MASU は Java を用いて実装されており，現在，規模はファイル数は 460，行数は約 99,000 行（ANTLR による自動生成部分が約 41,000 行）である．オブジェクト指向言語のソースコードを入力とし，対象ソースコードに含まれるクラスやメソッドなどのメトリックス値やソースコード解析結果を出力する．

現在，MASU は解析対象として Java と C#に対応している．Java に関しては Annotation を除くすべての要素の情報を解析できる．C#に関してはクラスやメソッドなどのオブジェクト指向言語共通の要素の解析を行うことができる．ただし，C#に関しても構造体（クラスと意味的に等しい）やプロパティ（メソッドと意味的に等しい）のような，メトリックス計測の際にオブジェクト指向言語共通の要素とみなす必要があると考えられる特有要素は解析できる．

MASU のアーキテクチャを図 1 に示す．本ツールはメインモジュールとプラグインから構成される．メインモジュールは更にソースコード解析部，プラグイン制御部，メトリックス集計部から構成される．

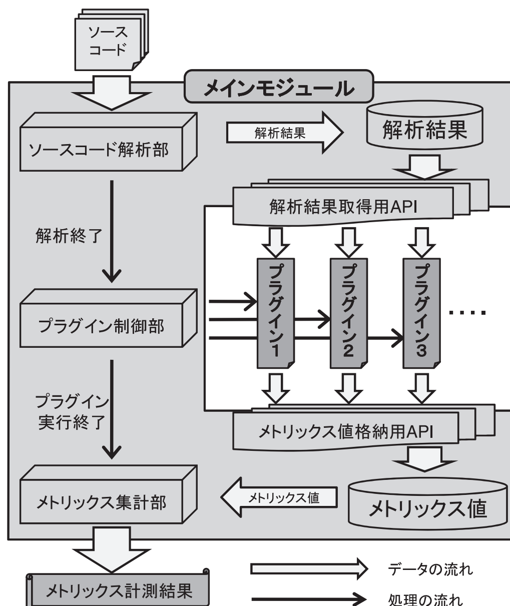


図 1　MASU の構成図
Fig. 1　Architecture of MASU.

メインモジュールはソースコード解析部でソースコードの情報を抽出し言語非依存なデータを構築する．次に，メトリックス制御部が指定されたプラグインを実行する．計測されたメトリックス値はメトリックス集計部で集計され，ユーザに提供される．

プラグインはメインモジュールが構築したデータを用いて個別のメトリックス値を計測する．

3.2 で述べた設計・実装方針は主にソースコード解析部とプラグインに適用されている．以降では，ソースコード解析部とプラグインの詳細，及び，MASU が提供するセキュリティマネージャについて説明する．

4.1 ソースコード解析部

ソースコード解析部は AST（Abstract Syntax Tree：抽象構文木）構築部と AST 解析部，データ構造構築部に分けられる．図 2 にソースコード解析部の構成と処理の流れを示す．

4.1.1 AST 構築部

入力として与えられたソースコードをもとに言語非依存な AST を生成する．図 2 に示されてある三つのソースコードは意味的には同じであるが，異なるプログラミング言語（Java，C#，Visual Basic）で記述されているため，構文的には異なっている．AST 構築部ではこのような各言語間の構文的な差異を吸収し，図 2 の右部に示してあるような共通の AST に変換する．AST 構築時に言語の記述様式の差異が吸収されるため，AST 解析部及びデータ構造構築部の大部分は様々な言語の解析に共通して利用することができる．

言語非依存な AST 構築の方針を以下に示す．

（方針 1）　AST 構築部は一つの大きな AST を構築するが，構築された AST は AST 解析部において複数の部分 AST とみなして解析される．例えば，図 3 の左上部の AST(Java) は図下部の部分 AST 集合とみなして解析される（AST(Java) はコンストラクタ定義を表す）．このため，AST 構築部では AST 全体を言語非依存にするのではなく，部分 AST が言語非依存になるように構築する．

部分 AST はデータが構築される要素ごとに定義され，構築される要素の内容を示すノード（図 3 の場合，CTOR_DEFINITION，PARAM_DEF など）が部分 AST のルートノードとなる．各要素間の所有関係は部分 AST のルートノードの位置関係で表される．具体的には，要素 A が要素 B を所有することを

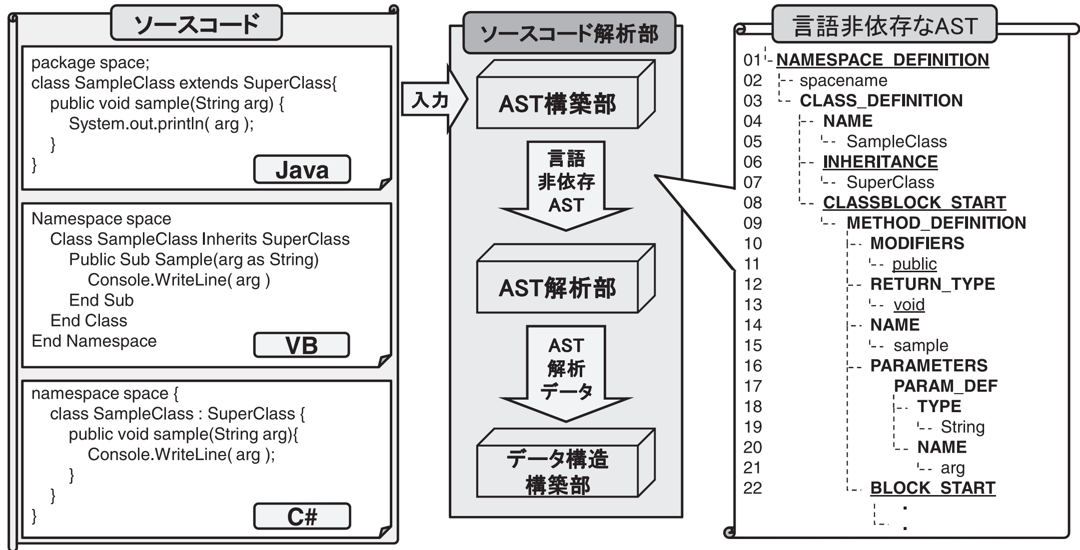


図 2 ソースコード解析部の構成
Fig. 2 Architecture of source code analysis unit.

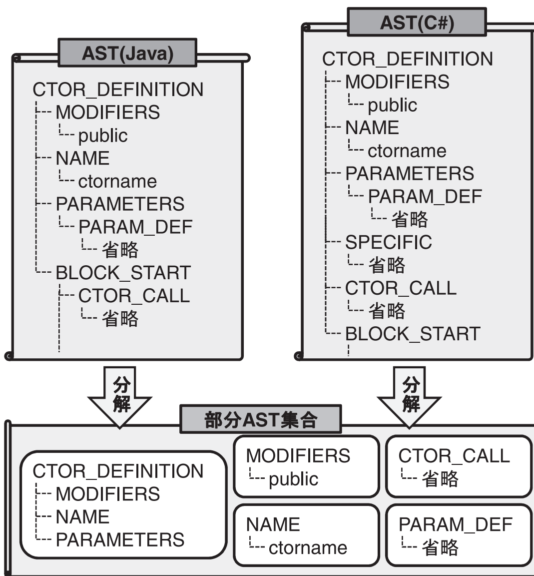


図 3 部分 AST 集合の例
Fig. 3 An example of a set of partial ASTs.

$inclue(A, B)$，要素 A の部分 AST のルートノードを $Root(A)$，ノード N_1 がノード N_2 の祖先ノードであることを $Ancestor(N_1, N_2)$ で表すとき，次の条件を満たすように AST が構築される．

$$include(A, B) \Longrightarrow Ancestor(Root(A), Root(B))$$

これにより，AST 全体の構造は異なっていたとしても，部分 AST が言語非依存であれば同一の AST として解析できる．例えば，図 3 の AST(Java)，AST(C#) はともにコンストラクタ定義を表すが AST(C#) ではコンストラクタ呼出しを表す部分 CTOR_CALL が “{}” を表す BLOCK_START ノードの子ノードではない（C#では同一クラスのコンストラクタ呼出しは “{}” の外側に記述されるため AST(C#) のようになる）．しかし，部分 AST は AST(Java)，AST(C#) ともに等しく，部分 AST の頂点ノード間の位置関係も同じであるため，解析時に全体的な構造の差異は影響しない．

また，このように AST を部分 AST の集合とみなして構築することにより，次の利点をもつ．

- 言語非依存な共通 AST を定義し，構築することは容易ではない．しかし，AST 全体の共通化に比べると，部分 AST の共通化は比較的容易に行える．
- 対応言語の追加が容易になる．対応言語を追加する際，AST 全体が共通化されていると，AST 全体の仕様を理解した上で AST 構築部を実装しなければならない．部分 AST の集合とみなすことにより，各部分 AST ごとの仕様を個別に理解すればよい．部分 AST 間の関係はルートノードの位置関係のみを把握すればよい．例えば，Visual Basic のようにコードブロックの開始を表すトークンがないため，

BLOCK_START ノードが AST 上に存在しない場合でも，コンストラクタ定義を表す AST 全体の構造を考慮して BLOCK_START ノードを追加するという作業を行う必要はない．

- 言語特有の情報を AST 上に残すことができる．部分 AST 間の関係はルートノードの位置関係のみを把握すればよいため，図 3 の AST(C#) のように言語特有の AST 構造をある程度残すことができる．また，部分 AST にはデータ構築に不要なノードは含まれないため，図 3 の AST(C#) の SPECIFIC ノードのような言語特有の要素を表すノードを AST 上に残すことができる．これらの情報は特定の言語に特化した解析には有用である．

（方針 2） 同一の概念を表す部分 AST が言語間で異なる場合，定義できる情報の量や種類が多い方を基準にして AST を定義する．例えば，Java の場合，一つのファイル内で一つの名前空間 (package) しか宣言できないが，C#の場合，複数の名前空間 (namespace) を定義することが可能であり，名前空間の階層構造も定義できる．つまり，C#の AST で Java の名前空間を表すことはできるが，Java の AST で C#の名前空間を表すことはできない．このような場合，AST 構築部は C#の AST を基準にして名前空間を表す部分 AST を構築する．

（方針 3） 部分 AST の構造は異なるが，定義できる情報の量や種類が等しい場合は Java の AST を基準にする．Java は現在存在する様々なプログラミング言語の中で開発者の関心が最も高く [27]，また，現在も継続的に言語機能の拡張が行われている．このため，Java の解析に関する拡張や変更を他の言語よりも行いやすくしている．

（方針 4） 言語特有の概念は，その言語の AST を基準として部分 AST を構築する．意味的に等しいオブジェクト指向共通の要素が存在する場合でも，共通の要素を表す部分 AST を構築することによる意味的な差異の吸収は行わない．これは AST 構築時に言語特有の概念を表す部分 AST をオブジェクト指向共通の要素を表す部分 AST に変換すると，特定の言語に特化した解析が困難になるためである．言語特有の概念をオブジェクト指向共通の要素とみなして，メトリックスを計測する場合は，AST 解析部で言語非依存な共通データを構築する際に意味的な差異を吸収する．

上記の方針に基づいて言語非依存な部分 AST を構築する際，言語差異を吸収のために行う処理の具体例を述べる．

［共通ノードの定義］ ソースコード上で意味的に等価な予約語に対して共通の AST ノードを定義する．例えば，図 2 の三つのプログラミング言語のソースコードにおいて，継承関係はそれぞれ “extends”，“:”，“inherits” で記述されているが，AST 上では共通ノード “INHERITANCE”（AST6 行目）で表す．図 2 の AST において，下線付きノードが共通ノードである．

［要素定義ノードの埋込］ AST の各ノードがソースコード上のどの要素に対応するかを示すノードを埋め込むことにより，記述順序の差異を吸収する．例えば，図 2 のソースコードが示すように，Java や C#の変数は先に型を宣言し，後に変数名を宣言するが，Visual Basic では先に変数名を宣言する．AST 上では “TYPE”（AST18 行目）や “NAME”（AST20 行目）のようにソースコード上のどの要素に対応するかを示すノードを埋め込むことにより，その子ノードである識別子が何を意味しているのかを特定する．図 2 の AST において，太字のノードが要素定義ノードである．

［ノードの移動・削除］ AST のノードの移動や削除を行うことにより，ノード間の関係の差異を吸収する．例えば，通常 Java の AST において名前空間を示すノードは，その名前空間内のクラスを示すノードと兄弟関係をもつが，C#の場合は親子関係をもつ．このような場合，Java のクラスを示す部分木を，その頂点ノードが名前空間を示すノードと親子関係をもつ位置に移動することにより，ノード間の関係の差異を吸収する．

4.1.2 AST 解析部

AST を解析し言語非依存な共通データを構築する．

図 4 が示すように，AST 解析部は複数の解析器と構築中データ管理部で構成される．解析器はソフトウェアを構成するそれぞれの要素に対して一つずつ用意される．各解析器は **4.1.1** で説明した部分 AST 一つを入力とし，部分 AST に対応する要素の情報のみを解析する．

各要素間の関係は，構築中データ管理部を通して，他の解析器と連携しながら解析する．

例えば，メソッド情報解析器はメソッドの情報のみを解析する．具体的には，変数情報解析器や識別子情報解析器と連携し，メソッドの引数やメソッド名などの情報を抽出する．また，構築中データ管理部から構築中のクラス情報を取得し，解析対象のメソッドを宣

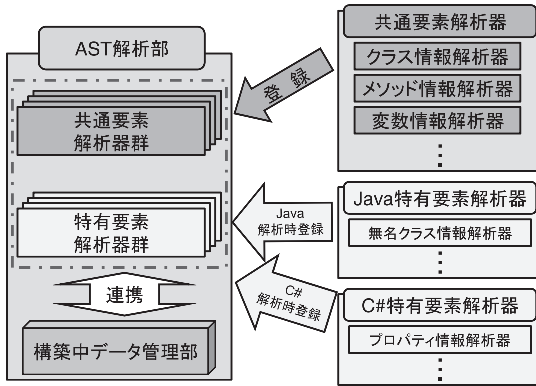


図 4　AST 解析部の構成
Fig. 4　Architecture of AST analysis unit.

言しているクラスとの所有関係を解析する．

異なる言語間で共通している要素の構文的な違いはAST 構築部で吸収されるため，共通要素に対する解析器は言語が異なっても正常に動作する．これにより，多言語対応に必要とするコストを大幅に削減することができる．

また，AST 解析部を複数の解析器で構成することにより，次のような解析を実現できる．

［計測するメトリックスに応じたソースコード解析］ AST 解析部に登録する解析器を変更することにより，容易に解析対象要素の変更が行える．より深い解析を行うために解析する要素を追加する場合は，その要素に対応した解析器を作成，登録するだけでよく，既存のコードを変更する必要はない．不要な要素の解析を行いたくないときは，その要素に対応した解析器を無効にすればよい．例えば，CK メトリックスの計測には，制御フローの情報は必要ないため，制御フロー情報解析器を無効にすれば，ソースコード解析のパフォーマンスが向上する．

［言語特有要素の解析］　AST 解析部は言語特有の要素を，図 4 のように言語特有要素解析器を用いて，意味的に等価な言語共通要素とみなして AST を解析する．例えば，C#のプロパティは意味的にはアクセサメソッドと等しいため，MASU のプロパティ解析器はプロパティの AST を解析し，意味的に等価なメソッドの情報を構築する．これにより，意味的には等しいが異なる要素の情報を統一的に扱える．

特定の言語に特化した解析を行う必要がある場合は，特有要素を特有要素として解析する解析器を登録することにより実現できる．しかし，メトリックスの多くはオブジェクト指向言語共通の概念に対して定義される．また，MASU の目的は共通の概念に対して定義されたメトリックスを，異なる言語で記述された複数のソフトウェアから統一的に計測することであるため，特有要素を特有要素として解析する解析器は現在未実装である．

4.1.3　データ構造構築部

AST 解析部で解析した情報をもとに，言語非依存な共通データを構築する．ここで言語非依存な共通データとは複数の言語で一般的に利用されるようなソフトウェアの要素に対するデータを意味する．

MASU が計測対象としているメトリックスは，ソースコードに記述されるソフトウェアの要素に対して定義され，その計測にソースコードに記述される様々な要素の情報を必要とする．このようなメトリックスの計測に関して広く有効なデータを提供するには，ソースコードに記述される要素に関する情報の粒度が細かく，精度が高いほど望ましい．データ構造構築部で構築されるデータが提供する情報の粒度や精度は基本的に次の方針で決定されている．

［粒度］　プログラム言語の文法は EBNF 記法などを用いて，プログラムの要素ごとに生成規則が厳密に定義されている．MASU はプログラミング言語の文法に生成規則が定義されている要素ごとにデータを構築する．ただし，構築されるデータにはファイルなどのようにプログラムの文法上に現れない要素も一部存在する．

［精度］　生成規則から得られる情報と等価な情報を得られるデータを構築する．要素の使用（変数参照やメソッド呼出しなど）を表す識別子に関しては，生成規則上では文字列情報しか得られないが，意味解析を行うことにより，要素の使用を表すデータ（要素の定義を表すデータと依存関係をもつ）を構築する．

ソースコードにはプログラミング言語の文法に定義されている内容以外は記述できない（コードコメントなど一部の例外は存在する）．このため，上記のように文法に定義されている要素ごとにデータを構築することにより，精度の高い情報を細かい粒度でソースコードから取得することができる．

また，言語拡張が行われたときメトリックス計測に必要な情報が増えたり，新しく追加された要素に対してメトリックスが定義される可能性がある．多くの場合，言語拡張は既存の生成規則を変更することなく，生成規則を追加することによって実現される．このた

め，上記の方針に従うことにより，既存の実装をそのまま利用し，追加された生成規則に対する実装のみを追加することにより，言語拡張に対応できる．

上記の方針でデータを構築する際，複数のプログラミング言語の文法に共通して定義されている要素（クラス，メソッド，文，式など）の情報を，言語非依存な共通データとして提供する．

4.2 プラグイン

メインモジュールは，メトリックス計測のためのデータ構造を提供し，個々のメトリックス値を計算するためのメトリックス計測ロジックはプラグインとして記述される．これにより，ソースコード解析部とメトリックス計測部が分離される．

各プラグインは基本的に，一つのメトリックスを計測するように実装される．個々のプラグインはメインモジュールが提供する解析結果取得用 API を用いて，メインモジュールが構築したデータを取得する．そして，計測対象要素のメトリックスを計測し，メトリックス値格納用 API を用いて計測結果を登録する．

各プラグインの作成手順を図 5 に示す．MASU はメインモジュールと協調するための API をもった抽象クラスを提供する．ユーザはメトリックスの計測対象となる要素に対応した抽象クラスのサブクラスを実装すればよい．現在，MASU がメトリックス計測対象としている要素はファイル，クラス，メソッド，フィールドである．

例えば，CK メトリックスを計測する場合は計測対象となる要素はクラスであるので，AbstractClassMetricPlugin クラスのサブクラスを実装する．メトリックス計測用のメソッド（measureClassMetric メソッドなど）にメトリックス計測ロジックを記述し，必要に応じてメインモジュールと協調するためのメソッドをオーバーライドすればよい．

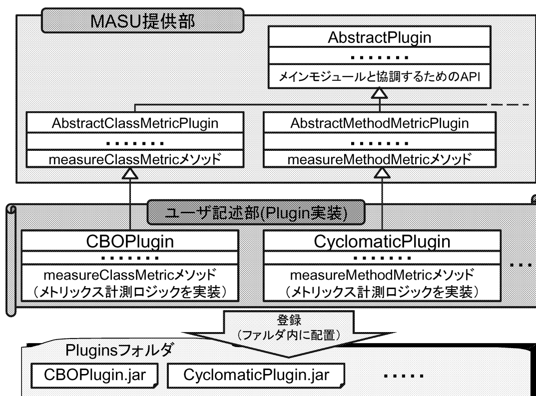


図 5 プラグイン作成の概略
Fig. 5 Outline of plug-in development.

作成された Plugin クラスはアーカイブ化して専用の plugins フォルダに配置すればメインモジュールが自動的にプラグインを認識し，プラグイン制御部で使用可能になる．

4.3 セキュリティ

メインモジュールが提供するデータは複数のプラグインで共有される．このため，プラグインが実行する処理によりメインモジュールが提供するデータが不正に変更されると，後に実行されるプラグインによって計測されるメトリックスを正しく計測できなくなるおそれがある．

このような問題を回避するため，MASU はそれぞれのプラグインを異なるスレッドで実行し，メインモジュールが提供するデータや各プラグインのデータへのアクセス制御をスレッド単位で動的に行うセキュリティマネージャを提供する．セキュリティマネージャはメインモジュールが構築したデータ構造をユーザ（プラグイン側）が勝手に変更できないよう管理している．また，あるプラグインが他のプラグインの情報を不正に変更できないようにしている．具体的には，次の処理を行う．

- 各プラグインの実行時のみ，実行中のプラグインスレッドに許可される権限の管理．
- 他のプラグインスレッドに不正に譲渡されないスレッド単位での権限管理．
- メインモジュール，プラグイン，GUI に共通して許可される権限の管理．
- 各プラグインのファイルシステムへのアクセス制御．

5. 評 価

5.1 設計・実装方針の効果

MASU は **4.1** で述べた設計・実装を行うことにより，多言語対応に必要とするコストを削減している．MASU の設計・実装の有効性を評価するために，筆者が多言語対応の実装を行う際に必要とした開発労力を評価した．実装は筆者が大学院在籍（博士前期課程）時に行った．実装当時の筆者のプログラミング経験は 2.5 年間であり，MASU の開発参加期間は 0.5 年である．また，AST を構築する際に必要となる構文解析

器は ANTLR を用いて生成した．ANTLR は入力となる構文定義ファイルに基づいて，構文解析器を自動生成するツールである．ANTLR の構文定義ファイルは既存のもの [1] を利用した．

5. 1. 1 言語非依存 AST 及び共通要素解析器の有効性評価

言語非依存の AST を構築し，共通要素解析器を再利用可能にしたとしても，言語非依存の AST の実装に必要とする開発労力が大きすぎるようであれば，言語非依存 AST 及び共通要素解析器の効果は低くなる．

そこで，我々は C#のソースコードから言語非依存 AST を構築し，言語共通要素を解析する機能を実装するために必要としたコストを計測した．共通要素解析器は Java のソースコードを解析するために実装した解析器を再利用した．その結果，Java を解析する機能の実装には共通要素解析器を新規実装する必要があったため，6 人月以上のコストを必要としたが，C#を解析する機能の実装に必要としたコストは，3 人日であった．

このことより，言語非依存 AST 及び共通要素解析器の有効性は高いといえる．

5. 1. 2 AST 解析部の設計及び共通要素解析器の有効性評価

AST 解析部は高い拡張性を提供することにより，共通要素解析器などの既存のコードに変更を加えることなく特有要素の解析を可能にしている．しかし，AST 解析部が高い拡張性を提供したとしても，共通要素解析器の実装に必要とする開発労力に比べて，特有要素解析器の実装に必要とする開発労力が大きすぎるようであれば，AST 解析部に拡張性の高い設計を適用し，共通要素解析器を再利用することの効果は低い．

そこで，我々は AST 解析部の設計及び共通要素解析器の有効性を示すために，オブジェクト指向言語共通要素の解析と Java 特有要素の解析に必要とした開発労力をクラス数を用いて評価した．結果を表 1 に示す．表 1 の列「解析器」は **4. 1. 2** で述べて AST 解析部を構成する解析器群の実装に必要としたクラスの数，「情報保存」は解析した情報を保存するためのクラスの数，「その他」はその他の処理に必要としたクラスの数を示す．

表 1 より，少なくとも Java 特有要素の解析にかかる実装はオブジェクト指向言語共通要素の解析にかかる開発労力に比べ非常に小さいことが分かる．このことより，AST 解析部の設計及び共通要素解析器の有効性は高いといえる．

表 1 オブジェクト指向言語共通要素と Java 特有要素の解析に要した開発労力（クラス数）

Table 1 The number of classes that were implemented for analyzing common elements of Object-Oriented Language and unique elements of Java.

対象要素	解析器	情報保存	その他
オブジェクト指向共通要素	118	157	19
Java 特有要素	15	5	7

5. 1. 3 考　察

Java と C#のオブジェクト指向共通要素に関する言語間の比較と，言語間の差異を吸収するために必要とした労力に関して考察する．同時に，他言語への対応に必要な労力に関しても定性的に評価する．

- 対応言語の追加を容易にするため，MASU は部分 AST が言語非依存になるように AST を構築している．AST の構築自体は，AST の構造に関するルールを構文定義ファイルに記述することで，ANTLR など構文解析器生成系が自動的に行う．構文定義上で各要素の生成規則は他の要素を参照する．ある要素 PE を表す部分 AST が要素 $CE_1 \sim CE_n$ に対応する n 個のノードで構成されるとき，要素 PE の生成規則上で要素 $CE_1 \sim CE_n$ が直接参照されていると，要素 PE を表す部分 AST を共通化するためのルールが記述しやすい．筆者らが C++，Visual Basic の文法に関して調査した結果，少なくともオブジェクト指向共通要素に関してはどちらの言語も，ほとんどの要素の生成規則上で部分 AST の構成ノードに対応する要素が直接参照されていた．このことから C++，Visual Basic ともに，部分 AST を言語非依存にする労力は小さいと考えられる．
- C#は一部のアクセス修飾子やブロックの開始・終了を表す予約語などが Java と一致しているが，継承関係やクラスのインポートを表す予約語は Java とは異なる．このように，意味的に等しい予約語には **4. 1. 1** で述べた共通ノードを割り当てる必要があったが，予約語に対する AST ノードの割当は構文定義ファイルに記述することで，ANTLR が自動的に行うため容易に実装することができた．このことから，C#のように頻繁に記述される予約語が Java と同じであるためソースコードが視覚的に似ている言語であっても，Visual Basic のようにソースコードが視覚的に異なる言語であっても，視覚的な差異を吸収し，言語非依存な AST を構築する労力はどちらも小さいと考えら

れる．

- C#のプロパティや構造体などは，Java には明示的には存在しないが，意味的には Java のメソッドやクラスと等しい．このため，プロパティや構造体の情報は，それと等価なメソッドやクラスを表すデータとして構築した．これを実現するためには，**4.1.2** で説明した解析器を新規実装する必要があった．また，プロパティの呼出しは，フィールドの参照・代入と区別することができないため，データ構造構築部にもプロパティを考慮した意味解析のロジックの実装した．

意味解析のロジックに影響を与える差異は，AST の構築段階では吸収することができなかった．これらの作業に必要とした労力は，共通ノードの割当や AST の構造を変更する労力と比較するとかなり大きかった．このことから，C#のようにソースコードが視覚的に似ている言語であっても，メトリックスを計測する際に言語共通の要素とみなして計測することが望ましいと考えられる要素をもっている言語や，特殊な意味解析を必要とする言語への対応は大きな労力を必要とすると考えられる．ただし，オブジェクト指向共通要素に関しては，意味解析のロジックも再利用できる場合が多いため，ソースコード解析器を個別に作成する労力に比較すると遥かに小さい労力で他言語に対応できると考えられる．また，言語特有要素に関する意味解析のロジックに関しても，言語を 1 対 1 で比較すると再利用できないが，複数の言語を考慮するとある程度再利用できる．例えば，Visual Basic において引数なしの関数呼出しは関数名の後に “()” を記述する必要がないため，フィールドの参照と区別することができない．しかし，これに対処するための意味解析ロジックは既に実装されている C#のプロパティ呼出しに対する意味解析ロジックが再利用できる．また，C#の特有の概念であるパーシャルクラスに対処するための意味解析ロジックは，C++のメソッドの非インライン定義に対処するために利用できると考えられる．このように Java では利用されない C#特有の意味解析ロジックも，C++や Visual Basic などの他言語を考慮すると，再利用することが可能であり，開発労力の削減に貢献すると考えられる．

5.2 実行性能

MASU の実行性能を評価するため，実際にソースコード解析を行い，使用した時間とメモリ空間を調査した．

具体的には，CPU が Pentium4 CPU 3.00 GHz，メモリ容量が 2 GByte，OS が Windows XP の PC を利用して，次の二つのソースコードを解析した．

［Java Platform Standard Edition 6］ 解析対象の開発言語は Java，規模はファイル数が 3,852，ソースコードの行数が約 12 万行，であった．解析に要した時間は約 392 秒，メモリ空間の使用量は最大約 838 MByte であった．

［CSgL］ 解析対象の開発言語は C#，規模はファイル数が 61，ソースコードの行数が約 3 万行，であった．解析に要した時間は約 59 秒，メモリ空間の使用量は最大約 51 MByte であった．CSgL は Java Platform Standard Edition 6 と比較すると小規模であるが，実装には C#特有の要素（構造体，プロパティ，#if などの前処理命令，as 演算子，C#特有の型，など）が多く使用されている．MASU はこれらの特有要素をオブジェクト指向共通の要素として解析することができた．

5.3 MASU を用いたプラグイン実装

MASU を用いたメトリックス計測プラグインの試作として，CK メトリックスとサイクロマチック数を計測するプラグインを実装した．

5.3.1 プラグインの規模

各メトリックス計測プラグインの規模と実装に要した時間を表 2 に示す．いずれのプラグインも実装は 1 人で行われた．表の列「メトリックス」は計測されるメトリックス名を，「総行数」は実装されたプラグインの行数（カッコ内は空白・コメント込みの総行数）を，「ML 行数」はメトリックス計測ロジックの行数を，「時間」は実装に要したおおよその時間を示す．また，WMC はすべてのメソッドの複雑度が一律であると仮定して実装した．

5.3.2 プラグイン実装例

MASU のプラグイン実装の例を図 6 に示す．

表 2　試作プラグインの規模と実装に要した時間
（総行数の括弧内は空白・コメント込みの行数）
Table 2　Size of sample plugins and time taken to implement them.

メトリックス	総行数	ML 行数	時間（分）
WMC	31 (74)	2	10
DIT	35 (81)	8	20
NOC	36 (73)	1	10
CBO	61 (121)	29	20
RFC	56 (117)	7	15
LCOM	114 (221)	48	40
サイクロマチック数	52 (115)	21	25

```
public class RfcPlugin extends AbstractClassMetricPlugin {
  @Override
  protected Number measureClassMetric(TargetClassInfo targetClass) {
    // この数が RFC
    final Set<MethodInfo> rfcMethods = new HashSet<MethodInfo>();

    // 現在のクラスで定義されているメソッド
    final Set<TargetMethodInfo> localMethods = targetClass.getDefinedMethods();
    rfcMethods.addAll(localMethods);

    // localMethods で呼ばれているメソッド
    for (final TargetMethodInfo m : localMethods) {
      rfcMethods.addAll(m.getCallees());
    }

    return new Integer(rfcMethods.size());
  }

  @Override
  protected String getDescription() {
    return "Measuring the RFC metric.";  //プラグインの簡易説明
  }

  @Override
  protected String getDetailDescription() {
    return DETAIL_DESCRIPTION;  // プラグインの詳細説明
  }

  @Override
  protected String getMetricName() {
    return "RFC";  // メトリックス名
  }

  @Override
  protected boolean useMethodInfo() {
    return true;  // このプラグインがメソッドに関する情報を利用するかどうか
  }

  @Override
  protected boolean useMethodLocalInfo() {
    return true;  // このプラグインがメソッド内部の情報を利用するかどうか
  }

  static {
    DETAIL_DESCRIPTION = "プラグイン詳細説明";
  }
}
```

メトリックス計測ロジックを実装

メインモジュールと協調するためのAPIをオーバーライド

図 6　RFC 計測プラグイン実装例
Fig. 6　An example of implementation for the plug-in that measures RFC.

図 6 は CK メトリックスの一つである RFC を計測するためのプラグインを実装している．一般に，RFC を計測するには，各クラスに定義されているメソッドと，それらのメソッドの中で呼び出されているメソッドの情報を抽出しなければならず，大きな開発労力を必要とする．しかし，MASU のプラグインとして実装することにより，表 2 や図 6 に示す程度のソースコードで実装することができる．図 6 から分かるように，メインモジュールと協調するための API は 1 行程度で実装することができ，プラグイン作成に必要な開発労力は実質的にはメトリックス計測ロジックを記述するメソッドの実装のみである．メトリックス計測ロジックの記述もメインモジュールが提供する情報を利用することにより，低コストで実装することができる．

また，WMC のように厳密な定義がされていないメトリックスを計測する場合，ユーザによって要求する定義は異なるであろうし，様々な定義を用いて計測結果を比較したいという要求も考えられる．MASU を利用すれば，図 6 のように measureClassMetric メソッド（クラスのメトリックスを計測する場合）を自由に実装することにより，ユーザ独自のメトリックス計測ロジックを用いてメトリックス計測を行うことができる．

6. 関 連 研 究

6.1　バイトコード解析ツール

Soot [26]　McGill 大学のプロジェクトとして開発されている Java バイトコード最適化フレームワーク．最適化のための情報としてバイトコード解析情報を提供するため，プログラム解析ツールとしても利用できる．
WALA [29]　IBM Research が開発した Java バイトコード解析ライブラリ．
バイトーコードを解析することにより，実際に実行されるコードの情報を得ることができるが，制御文の種類などのようにソースコードからでなければ得られない情報も多く存在する．また，バイトコードはコンパイラが生成したコードであり，人間が保守しなければないらないのはソースコードである．そして，メトリックスが示すのは，人間が作成したプロダクトの品質評価や，人間が行う保守のコスト予測などに使われる尺度である．ゆえに，メトリックスの計測は，バイトコードの特徴ではなく，ソースコードの特徴を測ることに意味があるといえる．

また，Soot や WALA は Java のバイトコードのみを対象としており，多言語対応はされていない．

6.2　ソースコード解析ツール

Ducasse らは言語非依存のリエンジニアリングプラットフォームとして MOOSE を開発している [4]．MOOSE は MASU と同様にソースコードを入力としたソフトウェアメトリックスの評価機能を提供する．しかし，MOOSE はソースコードを直接解析するわけではなく，CDIF や XMI などに変換された情報を解析するため，提供される情報はモデルベースの情報となる．モデルベースの情報はソフトウェアの視覚化などには有用であるが，ソースコードから直接得られる情報と比較すると情報量が減少する．また，ソースコードから CDIF や XMI への変換部はサードパーティーのツールに依存しており，MASU が提供するような拡張・変更容易性は備えていない．

Collard らはソースコードの情報を XML 形式で保

存するために srcML と呼ばれる XML の活用法を提案している [8]．srcML は C，C++，Java を対象としている．srcML から得られる情報は構文解析で得られる情報のみであり，意味解析で得られる情報は含まない．このため，構文情報を利用したコードコメントの管理などには有用であるが，意味解析で得られる情報を必要とするソフトウェアメトリックスの計測には利用できない．

Antoniol らは GCC (GNU Compiler Collection) が生成する AST を解析することにより，多言語に対応したソースコード解析ツール XOgastan を開発している [2]．XOgastan は原理的には GCC が対応しているすべての言語を解析できる．しかし，GCC が対応していない言語の解析を行うための設計上の工夫に関しては述べられていない．また，XOgastan は意味解析が不十分であるため，各変数の関数内における使用回数などに関する情報は取得できるが，関数内に記述されている式などの文脈に関する情報は取得できない．MASU の AST 解析部のような解析対象要素の追加を容易にするための設計に関しても述べられていない．

福安らは細粒度のソフトウェアリポジトリに基づいた CASE ツールプラットホームとして Sapid を開発している [10]．Chen らは C 言語を対象としたプログラムの情報の抽象化システムを開発している [5]．これらのシステムは MOOSE，srcML，XOgastan に比べ，細かい粒度でソフトウェアの要素を解析しているため，これらのシステムの有効性は高く，MASU と同じようにメトリックス計測ツールの開発基盤としても利用できる．ただし，MASU はメトリックス計測を行う上で，次の点に関して，これらのツールよりも洗練されている．

- 複数のプログラミング言語を統一的に扱う．
- 複数のメトリックスの計測に対応するために，提供するデータを容易に変更・追加できるよう設計されている．
- メトリックス計測ツールをプラグイン化している．

C/C++のソースコードを正確に解析するには，テンプレートの情報を解析する必要がある．しかし，テンプレートの解析は複雑であるため，C/C++を対象としたソースコード解析ツールの多くはテンプレートの解析に十分に対応していない．Gschwind らが開発している TUAnalyzer は，解析の前準備として GNU の C/C++コンパイラを利用することにより，テンプレートの解析を実現している [11]．MASU は将来 C/C++の対応を検討しており，実装には，TUAnalyzer 同様に解析の事前準備として GNU の C/C++コンパイラを利用することを考えている．

6.3 AST 変換技術

長谷川はあるプログラミング言語 G_1 のツールを他の言語 G_2 に適用可能にする手法を提案している [12]．この手法は言語 G_2 の AST を言語 G_1 の AST に変換することにより実現される．この手法により，特定の言語を対象とした既存のメトリックス計測ツールを他の言語に適用することが可能となる．しかし，同一の言語を対象としたツールであっても，そのツールが扱う AST はツールごとに異なる．このため，AST の変換ルールは各ツールごとに実装する必要がある．

MASU は既存のツールの再利用には応用できないが，**5.3** で示したようにメトリックス計測ロジックの記述のみであれば低コストで再実装することができる．AST 変換ルールの実装には構文解析の専門的な知識が要求されるため，AST 変換を用いるよりも，MASU を用いて再実装した方が低コストで等価なツールを用意できる可能性は高い．また，MASU のソースコード解析部で言語間の差異が吸収されるため，ツールごとに AST 変換ルールなどを実装する必要はない．つまり，MASU が言語 G に対応すれば，MASU のソースコード解析部を利用したすべてのツールが言語 G に適用可能となる．

7. むすび

本論文では，メトリックス計測プラグイン開発基盤 MASU について述べた．MASU は次の特徴をもっており，MASU を用いることによって，従来に比べて低いコストでソースコードからメトリックスを計測することが可能になる．

- 複数のプログラミング言語のソースコードに対して適用可能である．
- 多言語の解析から得た結果を統一的に扱える．
- 複数のメトリックスの計測に対応するために，提供するデータの追加や変更を容易に行える．
- 必要最低限のメトリックス計測ロジックを記述するだけで，新たなメトリックスを計測可能である．

MASU の開発は実用的な段階にまで到達しており，実際にいくつかの研究で活用されている [9], [15], [20], [21], [24]．

現在，MASU は Java 若しくは C#のソースコー

ドを入力とすることができる．今後，C++，Visual Basic などの他のオブジェクト指向言語にも順次対応していく予定である．また，解析対象をソースコード以外に広げることも考えている．例えば，実行履歴を解析して共通データを構築するような解析部を作成することで，動的なデータをもとにしたメトリックス計測が可能になる．

謝辞　本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．

文　　献

[1] ANTLR. http://www.antlr.org/

[2] G. Antoniol, M.D. Penta, G. Masone, and U. Villano, "XOgastan: XML-oriented gcc AST analysis and transformations," Proc. 3rd IEEE International Workshop on Source Code Analysis and Manipulation, pp.173–182, Sept. 2003.

[3] Aqris software. RefactorIT. http://www.aqris.com/

[4] A.L. Baroni and F.B. Abreu, "An OCL-based formalization of the MOOSE metric suite," Proc. 7th ECOOP Workshop on Quantitative Approaches in Object-Orietend Software Engineering, 2003.

[5] Y.F. Chen, M.Y. Nishimoto, and C.V. Ramaoorthy, "The C information abstraction system," IEEE Trans. Softw. Eng., vol.16, no.3, pp.325–334, March 1990.

[6] S. Chidamber and C. Kemerer, "A metric suite for object-oriented design," IEEE Trans. Softw. Eng., vol.25, no.5, pp.476–493, June 1994.

[7] Clarkware consulting inc. JDepend. http://www.clarkware.com/

[8] M.L. Collard, J.I. Maletic, and A. Marcus, "Supporting document and data views of source code," Proc. ACM Symposium on Document Engineering, pp.34–41, Nov. 2002.

[9] 枝川拓人，赤池輝彦，肥後芳樹，楠本真二，"画面遷移とデータベース処理を考慮したトランザクションファンクション識別手法の詳細化と実装，" 信学技報，SS2008-17, July 2008.

[10] 福安直樹，山本晋一郎，阿草清滋，"細粒度ソフトウェア・リポジトリに基づいた case ツール・プラットフォーム sapid，" 情処学論，vol.39, no.6, pp.1990–1998, June 1998.

[11] T. Gschwind, M. Pinzger, and H. Gall, "TUAnalyzer—Analyzing templates in C++ Code," Proc. 11th Working Conference on Reverse Engineering, pp.48–57, Nov. 2004.

[12] 長谷川勇，"AST 変換を用いた他言語へのツール適用，" 信学技報，SS2008-22, July 2008.

[13] B. Henderson-Sellors, Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1996.

[14] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," Proc. International Symposium on Applied Corporate Computing, pp.78–84, Oct. 1995.

[15] 堀　直哉，岡野浩三，楠本真二，"モデル検査技術を用いたインバリアント被覆テストケースの自動生成による Daikon 出力の改善，" ソフトウェア工学の基礎 XV，pp.41–50, Nov. 2008.

[16] instantiations inc. CodePro AnalytiX. http://www.instantiations.com/

[17] JavaCC. http://javacc.dev.java.net/

[18] JMetric. http://www.it.swin.edu.au/projects/jmetric/products/jmetric/default.htm

[19] R. Lincke, J. Lundberg, and W. Lowe, "Comparing software metrics tools," International Symposium on Software Testing and Analysis, pp.131–141, July 2008.

[20] 宮崎宏海，肥後芳樹，井上克郎，"アイテムセットマイニングを利用したコードクローン分析作業の効率向上，" 信学技報，SS2008-18, July 2008.

[21] 三宅達也，肥後芳樹，井上克郎，"ソフトウェアメトリクスとメソッド内の構造を用いたリファクタリング支援手法の提案，" 信学技報，SS2008-25, July 2008.

[22] 三宅達也，肥後芳樹，井上克郎，"メトリクス計測プラグインプラットフォーム MASU の開発，" ソフトウェアエンジニアリング最前線，pp.63–70, Sept. 2008.

[23] T. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol.2, no.4, pp.308–320, Dec. 1976.

[24] 村尾憲治，肥後芳樹，井上克郎，"ソフトウェアメトリクス値の変遷に基づいた注力すべきモジュールを特定する手法の提案，" 信学論（D），vol.J91-D, no.12, pp.2915–2925, Dec. 2008.

[25] P. Oman and S.L. Pleeger, Applying Software Metrics, IEEE Computer Society Press, 1997.

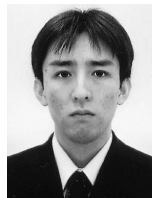
[26] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/

[27] TIOBE Software: The Coding Standards Company. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[28] Virtual machinery. JHawk. http://www.virtualmachinery.com/

[29] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page

（平成 20 年 10 月 31 日受付，21 年 3 月 9 日再受付）



三宅　達也

平 19 阪大・基礎工・情報卒．現在同大大学院博士前期課程 2 年．リファクタリングの研究に従事．

肥後　芳樹　（正員）

平 14 阪大・基礎工・情報中退．平 18 同大大学院博士後期課程修了．平 19 阪大・情報・コンピュータサイエンス・助教．博士（情報科学）．コードクローン分析やリファクタリングに関する研究に従事．情報処理学会，IEEE 各会員．

楠本　真二　（正員）

昭 63 阪大・基礎工・情報卒．平 3 同大大学院博士課程中退．同年同大・基礎工・情報・助手．平 8 同大講師．平 11 同大助教授．平 14 阪大・情報・コンピュータサイエンス・助教授．平 17 同学科教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．情報処理学会，IEEE，JFPUG 各会員．

井上　克郎　（正員）

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59〜昭 61 ハワイ大マノア校・情報工学科・助教授．平元阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．博士（工学）．平 14 阪大・情報・コンピュータサイエンス・教授．ソフトウェア工学の研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE，ACM 各会員．