情報処理学会論文誌　プログラミング　Vol. 1　No. 1　1234–1249 (Jan. 2008)

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

坂本一憲[†1]　大橋昭[†1]　太田大地[†2]
鷲崎弘宜[†1]　深澤良彰[†1]

近年，プログラミング言語の多様化に伴い，複数のプログラミング言語を利用したソフトウェア開発が普及している．ソフトウェア開発を支援するため，メトリクス測定を始めとする様々なソースコードの解析および変形ツールが開発されている．

しかし，これらの既存ツールの多くは 1 つのプログラミング言語を対象として開発されているため，以下で述べる 2 つの問題点がある．プログラミング言語とツール間に多対多の関係があり，全ての言語とツールの組み合わせに対して実装した場合，非常に莫大なコストが必要となる点，ツール毎に実装や仕様に差異が存在していて，複数のプログラミング言語で開発されたソフトウェアに適用しづらい点である．

本論文では，上述の問題を解決するために，複数のプログラミング言語に対応するソースコード処理フレームワーク UNICOEN（UNIfied source COde ENgineering framework）を提案する．UNICOEN はシンタックスに基づいて言語非依存な言語モデルを提供する．言語モデルは，プログラミング言語とツール間の多対多の関係を，言語と UNICOEN 間およびツールと UNICOEN 間の多対一の関係に簡略化する．このことは，ツールの開発コストを大幅に削減して，ツール間の差異を低減する．我々は，UNICOEN 上で開発した 7 種類のプログラミング言語に対応する 3 種類のツールを評価して，UNICOEN が上述の問題を解決することを確認した．

UNICOEN: A Unified Framework for Code Engineering Supporting Multiple Programming Languages

Kazunori Sakamoto,[†1] Akira Ohashi,[†1] Daichi Ota,[†2]
Hironori Washizaki[†1] and Yoshiaki Fukazawa[†1]

As programming languages continues to evolve and become more multi-faceted, software development with multiple programming languages becomes popular. To aid software development, many analysis and transform tools for source code such as metrics-measurement tools are being developed.

However, there are two problems because these tools support only one programming language. P1: Enourmous development costs are required to implement all conbinations between programming languages and tools because there is a many-to-many relationship between them. P2: It is hard to introduce the tools for software development with multiple programming languages because there are differences of implementations and specifications in the tools.

In this paper, we propose a framework for processing source code supporting multiple programming languages named UNICOEN (UNIfied source COde ENgineering framework). UNICOEN simplifies the many-to-many relationship between programming languages and tools to one-to-many relationships between programming languages/tools and UNICOEN. It drastically reduces development costs and prevents differences of implementations and specifications between tools. Conclusively, we evaluated UNICOEN developing 3 tools which supports 7 programming languages.

1. はじめに

近年，様々なプログラミング言語（以降，言語）が開発され，その多様化が進んでいる．例えば，1972 年に開発された歴史のある C 言語は現在でも広く普及している．その一方で，Kotlin[1] や Xtend[2] 言語など新しい言語が次々と生まれている．また，言語は改良され，言語仕様は変化し続けている．

様々な言語が存在する中で，ソースコードを処理するツール（以降，ツール）が存在する．ツールの処理内容は，大きく分けてソースコードの解析と変形の 2 つに分けられる．ソースコードを解析するツールとして，メトリクス測定ツールや静的解析ツール，また，ソースコードの解析に基づいて変形するツールとして，ソースコード整形ツールやアスペクト指向プログラミング（Aspect Oriented Programming; AOP）処理系が挙げられる．これらのツールは，ソフトウェアの品質や開発効率を向上させるツールとして注目を浴びている[3]．

現在，多くのツールが言語毎に開発されている．例えば，静的解析ツールの FindBugs[4] は Java 言語のみに，JSLint[5] は JavaScript 言語のみに対応している．また，AOP 処理系の AspectJ[6] は Java 言語のみに，AOJS[7] は JavaScript 言語のみに対応している．このように，言語とツールの間には多対多の関係がある．例えば，各ツールが 1 つの言語のみに

†1 早稲田大学 基幹理工学研究科 情報理工学専攻
Department of Computer Science and Engineering, Waseda University
†2 株式会社 ACCESS
ACCESS CO., LTD.

対応していて，全てのツールと言語の組み合わせを考えた場合，図 1 のように「言語の種類数」×「ツールの種類数」通りの実装が必要になる．しかし，言語とツールの間の多対多の関係は，以下で示す 2 つの問題点を引き起こす．問題 1：全てのツールが全ての言語に対応するために莫大な開発コストが必要である点，問題 2：対応言語が異なる同じ種類のツール間に差異が生じる点である．そのため，優れた言語やツールが存在するにもかかわらず，使用する言語によってはその恩恵が十分に得られない．

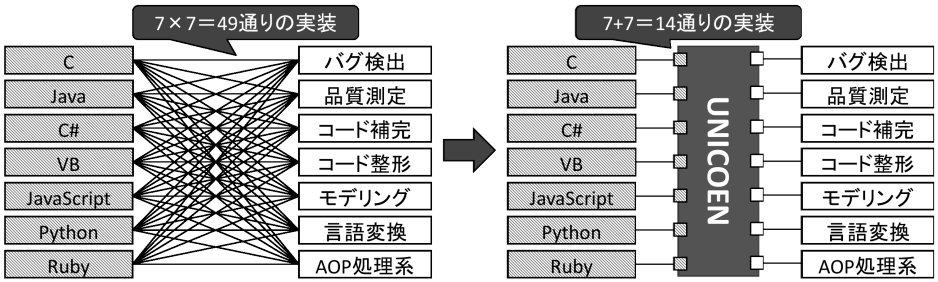


図 1　UNICOEN による言語とツールの多対多の関係の簡略化
Fig. 1　A simplification for the many-to-many relationship between programming languages and tools

本論文では，以上の問題点を解決するために，UNICOEN（UNIfied source COde ENgineering framework）[⋆1]を提案する．UNICOEN は，解決 1：言語非依存な統合コードモデルを提供して，解決 2：シンタックスに基づいてソースコードを統合コードモデル上のオブジェクトにマッピングする．その上で，解決 3：統合コードモデル上の汎用的な共通処理を 2 種類の API で提供する．これによって，図 1 のように言語とツールの多対多の関係を言語と UNICOEN 間およびツールと UNICOEN 間の多対一の関係に簡略化する．

我々は，UNICOEN 上で言語の人気ランキング（TIOBE Programming Community Index for January 2012[8]）上位 12 位中の 7 言語の対応を実装して，その上で，3 種類のツールを開発した．言語対応の拡張とツール開発コストが既存ツールと比較して少なく，開発したツールが言語間の差異を低減するため，問題 1 と問題 2 を解決できることを確認した．

⋆1 UNICOEN は IPA の 2010 年度未踏 IT 人材発掘・育成事業の支援を受けて開発した．その成果が認められ，著者らを含む開発メンバーは経済産業省および IPA から未踏スーパークリエータの認定を受けた．

表 1　UNICOEN が対象とするツールの一覧
Table 1　A catalog of targetted tools by UNICOEN

ツールの種類	シンタックスハイライト，シンタックスチェッカー	バグパターン検出ツール，静的解析ツール，統合開発環境，リファクタリングツール，メトリクス測定ツール，カバレッジ測定ツール	言語処理系，コンパイラ
意味解析への依存度	低	中	高

本論文の構成は以下のとおりである．2 節で既存のツールの問題点を説明して，3 節で UNICOEN の概要を示す．その上で，UNICOEN の詳細を 4 節で説明して，5 節で実際に開発したツールに基づいて UNICOEN の評価を行う．最後に，6 節にて関連研究について述べ，7 節にて本論文をまとめる．

2. 既存ツールの問題点

UNICOEN が対象とするツールの一覧を表 1 で示す．UNICOEN はソースコードを入力として解析結果を出力するツールや，解析結果に基づいてソースコードを変形するツールの開発を支援する．そのために，UNICOEN は完全なシンタックスの情報と一部のセマンティクスの情報に基づいてソースコードを構造化する．

対象とするツールはシンタックスの情報だけでなく，セマンティクスの情報も利用する．意味解析，つまり，セマンティクスの情報にどこまで依存するかによってツールを分類できる．ほとんどセマンティクスの情報に依存しないツールの依存度を低，完全にセマンティクスの情報に依存するツールを高，それ以外を中とした．例えば，シンタックスハイライトはシンタックスの情報のみに基づいてコードをハイライトする．コンパイラはソースコードをマシン語や中間言語に変換するために，完全なセマンティクスの情報を必要とする．メトリクス測定ツールはクラス構造などメトリクスが必要とするセマンティクスの情報のみを必要とする．依存度が低いツールは UNICOEN が提供する情報だけで開発できるが，高くなるほど開発者が UNICOEN 上で意味解析する処理を記述する必要性が上がる．

既存ツールにおける 2 つの問題点を以下で説明する．

2.1　問題 1：莫大な開発コスト

言語とツール間に多対多の関係があるため，全てのツールが全ての言語に対応するために必要な開発コストが莫大である．そのため，ある言語のみに対応したツールが開発されてから，長い時間を経て他の言語向けにツールが移植されたり，一部の言語に対して対応するツールが存在しないケースがある．したがって，ツール開発者は対応言語を拡張するために

多大な労力を割く必要がある．また，ツール利用者は，開発プロジェクトが採用する言語によってはツールの恩恵を受けられなかったり，ツールを利用するために採用可能な言語の選択肢の幅が狭まるといった問題が生じている．

例えば，C 言語に対応した静的解析ツール Lint[9] が 1977 年にリリースされてから，JavaScript 言語に対応した同様のツール JSLint が 2002 年に，Python 言語に対応した同様のツール Pylint[10] が 2004 年にリリースされている．なお，JavaScript と Python 言語はそれぞれ 1995 年，1990 年にリリースされている．Ruby 言語に対応した同様のツールは我々が調査した限りでは存在していない．このように，ツール利用者がツールのサポートを受けられる言語は一部のみであり，サポートが広がるまでに長い期間を要する．

言語の機能拡張や仕様変更に対応する保守コストが莫大であることも問題である．例えば，2008 年に Python 言語はバージョン 2 から 3 へアップデートしていて，その際，後方互換性が破棄された[11]．Python 言語バージョン 2 では print が特殊なステートメントとして扱われていたが，バージョン 3 では print がただの関数として扱われるようになったため，図 2 のような相違点が生じた．この変更によって，Pylint を含めた多くのツールの修正が必要となった．このように，単にツールが対応する言語を拡張するために，一度ツールの機能拡張を行えば良いというわけではない．機能拡張した後も対応言語に対するサポートを継続する必要があり，こうした保守作業を含めて莫大な開発コストが必要である．

```
1 print "for version 2"        # Pythonバージョン2でのみ動作
2 print("for version 2 and 3") # Pythonバージョン2と3の両方で動作
```

図 2　Python バージョン 2 と 3 の相違点
Fig. 2　A difference between Python version 2 and 3

2.2　問題 2：ツール間の差異

異なる言語に対応した同じ種類のツールを組み合わせる場合，ツール間の差異のために期待する結果が得られなかったり，組み合わせるために追加のコストが必要になったりする．そのため，複数の言語を利用するプロジェクトで複数のツールを組み合わせても，これらの言語で記述されたソースコードを対象としてツールを適用できない場合がある．

例えば，テストカバレッジ測定ツール EMMA[12] は Java 言語に，Code coverage measurement for Python[13] は Python 言語に対応している．これらのツールを用いてステートメントカバレッジを測定する際，前者は Java VM 上の命令単位で，後者はソースコードの行単位で測定する．サーバークライアントモデルにより，サーバーサイドを Java 言語，クライアントサイドを Python 言語で開発したソフトウェアを対象とする場合，これらのツールの測定基準が異なるため，測定結果を統合して総合的に評価することが難しい．

また，AOP 処理系の AspectJ は Java 言語に，AOJS は JavaScript 言語に対応している．同様にサーバーサイドを Java 言語で，クライアントサイドを JavaScript 言語で開発するソフトウェアにおいて AOP を利用する場合，Java と JavaScript 言語のソースコードに対して，それぞれアスペクトを記述しなければならない．アスペクトのモジュール性が低下する上，アスペクトの記述方法は AOP 処理系毎に異なるため，ツール利用者は各記述方法を学ぶ必要がある．全てのメソッド実行時にロギングを行う AspectJ と AOJS のアスペクトの例をそれぞれ図 4 と図 3 で示す．両者のアスペクトが同じ内容を示しているのにも関わらず，記述方法が大きく異なることが分かる．

```
1 public aspect Logger {
2   pointcut allMethod() : execution(* *.*());
3
4   before() : allMethod() {
5     System.out.println(thisJoinPoint.getSignature() + " is executed.");
6   }
7 }
```

図 3　メソッドの実行に対するロギングする AspectJ のアスペクト
Fig. 3　A logging code for executing methods in AspectJ

```
1 <?xml version="1.0" ?>
2 <aspectsetting>
3 <function functionname="/*" pointcut="execution">
4 <before>
5   <![CDATA[console .log ( __name__ + " is executed ."); ]]>
6 </before>
7 </function>
8 </aspectsetting>
```

図 4　メソッドの実行に対するロギングする AOJS のアスペクト
Fig. 4　A logging code for executing methods in AOJS

3. UNICOEN の全体像

UNICOEN の全体像を図 5 で示す．UNICOEN は統合コードモデルを中核に据え，汎用的な共通処理として，対応言語拡張者向け API とツール開発者向け API を提供する．

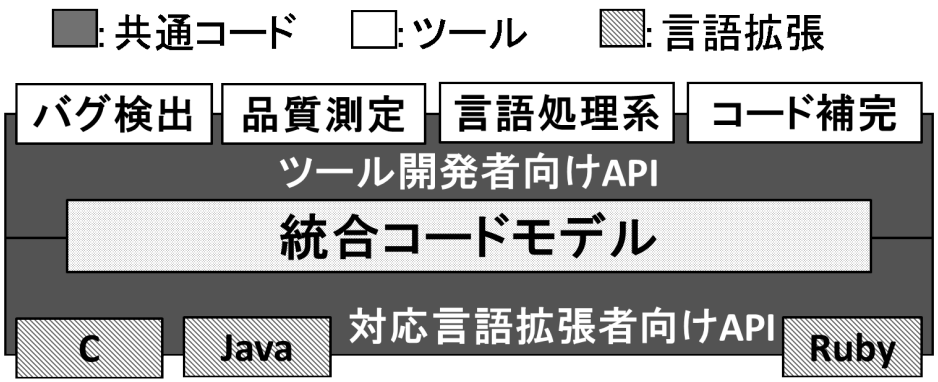


図 5　UNICOEN の全体像
Fig. 5　An overview of UNICOEN

UNICOEN は C# 4.0 で開発されており，.NET Framework および Mono 上で動作するフレームワークである．UNICOEN は，Apache2.0 ライセンスのもとでオープンソースソフトウェアとして開発されており，Github 上のプロジェクトサイトからソースコードをダウンロードできる[14)15)]．また，UNICOEN は NuGet パッケージとして公開しており，NuGet を利用して容易にインストールして使用できる[16)]．

UNICOEN は共通な抽象構文木（Abstract Syntax Tree; AST）の仕様として，解決 1：言語非依存な統合コードモデルを提供する．ソースコードから得られる言語非依存な抽象構文木のインスタンスを統合コードオブジェクトと呼ぶ．統合コードモデル上でツールを開発することで，問題 2：対応言語が異なる同じ種類のツール間に差異が生じる点を解決する．UNICOEN は，ツール開発者向け API として，W3C が標準化した DOM と類似した機能を提供する．具体的には，ソースコードと統合コードオブジェクトの相互変換，および，統合コードモデル上で解析や変形を行うため要素の抽出・追加・変更・削除機能を提供する．そのため，ソースコードを文字列処理によって処理のではなく，構造化されたモデル上で処理することで，文字列処理によって実装可能な機能を統合コードモデル上でより直感的に低コストで実装可能である．UNICOEN は，2 節で挙げたツールの開発を支援する．UNICOEN がツール開発者向け API を提供するために，ソースコードと統合コードオブジェクトを相互に変換する Object-code mapper（以降，OC マッパー）を言語毎に実装する必要がある．そのため，UNICOEN は，対応言語拡張者向け API として，OC マッパーの実装に必要な汎用的な共通処理を提供することで，対応する言語の拡張を支援する．なお，ある言語について OC マッパーを一度実装すれば，任意の処理ツールから利用できる．UNICOEN は，解決 2：シンタックスに基づいてソースコードを統合コードモデル上のオブジェクトにマッピングすることで，対応言語の拡張コストを低減する．さらに，解決 3：統合コードモデル上の汎用的な共通処理を 2 種類の API で提供することで，ツールの開発コストと対応言語の拡張コストの両方を低減する．以上から，問題 1：全てのツールが全ての言語に対応するために莫大な開発コストが必要である点を解決する．

利用する API の種類によって UNICOEN の利用者を分類することができ，ツール開発者向け API を利用してツールを開発する利用者（以降，ツール開発者）と，対応言語拡張者向け API を利用して UNICOEN が対応する言語を追加する利用者（以降，対応言語拡張者）の 2 種類に分けられる．ツール開発者は，統合コードモデル上でソースコード処理を記述することで，対応する言語を意識せずにツールを開発できる．UNICOEN を用いて開発したツールは，UNICOEN が対応する全ての言語に対応できる．一方，対応言語拡張者は，OC マッパーを開発することで，ツールを意識せずに対応言語を拡張できる．UNICOEN が対応した言語について，UNICOEN を用いて開発された全てのツールが対応できる．このようにして，UNICOEN は図 1 のように，対応言語とツール間の多対多の関係を対応言語と UNICOEN 間およびツールと UNICOEN 間の多対一の関係に簡略化する．

4. UNICOEN の詳細

本節では，統合コードモデル，対応言語拡張者向け API，ツール開発者向け API の 3 つに分けて UNICOEN の詳細を説明する．

4.1　統合コードモデル

UNICOEN は，言語非依存なツールの開発を実現するために，様々な言語で記述されたソースコードを構造化できるような統合コードモデルをクラスとして提供する．統合コードオブジェクトは統合コードモデル上のインスタンスであり，木構造を有している．そのた

め，クラスを表す統合コードオブジェクトは，子要素としてメソッドを表す統合コードオブジェクトを持ち，さらにその子要素として引数やブロックなどを表す統合コードオブジェクトを持つといったように，再帰的な構造を持つ．

統合コードモデルは主に構文解析に基づいてソースコードをオブジェクトとして構造化するため，完全な意味解析を必要としない．例えば，UNICOEN は二項式の構文を認識するが，演算子の意味（例えば，+ 演算子が加算か文字列結合か）まで解釈しない．したがって，UNICOEN は GCC や LLVM のようなコンパイラフレームワーク，Java VM や.NET Framework のような中間言語の実行環境とは異なる．これらの既存ソフトウェアとの詳細な相違点は 6 節にて示す．UNICOEN は，ソースコードを構造化することでツールの開発コストを削減して，意味解析を省くことで対応言語の拡張コストを低減する．

我々は，ソースコード中に現れるほぼ全てのトークンを記憶できるように統合コードモデルを設計した．しかし，一部のトークンは，統合コードオブジェクトを構築する上で除去される．例えば，(1+2)*3 という式がソースコード中に現れる場合を考える．この式を統合コードオブジェクトで表現すると，図 6 のような木構造になる．この例では，+ 演算子の方が * 演算子よりも深い位置に存在しており，+ 演算子の方が * 演算子よりも優先度が高いことが分かる．演算子の優先度を統合コードオブジェクトの深さで表現できるため，演算子の優先度を示す括弧をトークンとして記憶する必要がない．したがって，括弧は統合コードオブジェクトで記憶されない．このように，構文木が全てのトークンを記憶するのに対して，抽象構文木が一部のトークンを記憶するように，統合コードオブジェクトでは意味を損なわないようにいくつかのトークンを省いて記憶する．

我々は，C，Java，C#，Visual Basic，JavaScript，Python，Ruby の 7 種類の言語について，それぞれの文法の和集合を取ることで統合コードモデルを設計した．和集合を取る際に，各言語の意味論に基づいて類似する構文を共通要素として，そうでないものはそれぞれ異なる要素として扱った．

例えば，while 文の統合コードモデルを考える．図 7 のように，多くの言語において while 文は，ループの継続を判定する条件式，ループ内で実行する命令の 2 つの要素から構成される．しかし，Python 言語では while 文に else 節を付加できる．この else 節は，ループの継続を判定する条件式が偽になった際に，ループを抜ける直前で実行する命令を示す．したがって，上述の 7 種類の言語について和集合を取り，while 文の統合コードモデルは，ループの継続を判定する条件式，ループ内で実行する命令，ループを抜ける直前で実行する命令の 3 つの要素を持つように設計した．

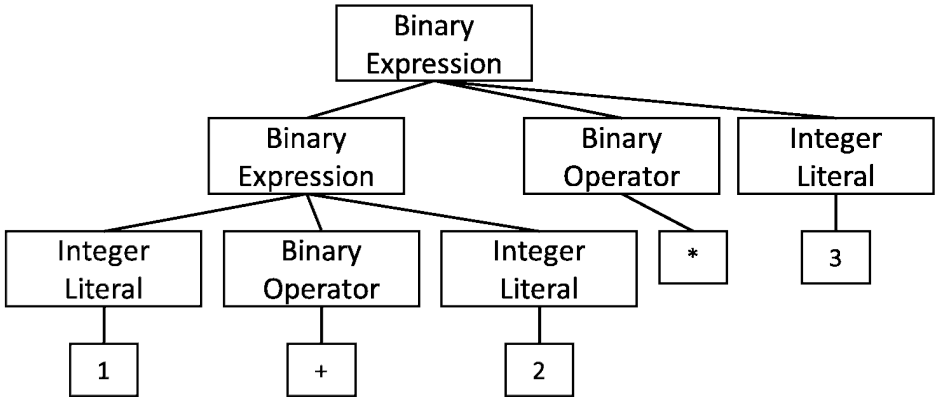


図 6　(1+2)*3 の統合コードオブジェクト
Fig. 6　A unified code object of (1+2)*3

Abstract Syntax DescriptionLanguage（ASDL）を用いて記述した統合コードモデルの定義の一部を図 8 に示す．なお，ASDL の全体を付録にて提示する．一般に，多くの手続き型言語では式（エクスプレッション）と文（ステートメント）を評価値を持つか否かで区別することが多い．しかし，Ruby 言語は，他の言語で文として扱われている構文の多くを式として扱っており，文はほとんど存在しない．例えば，Ruby 以外の 6 種類の言語では，if 文や while 文，関数定義は文であるが，Ruby 言語ではすべて式である．そこで，我々は文を式の特殊なケースと考えて，両者の区別をなくした．したがって，統合コードモデルでは文と式をどちらも式として扱う．このように和集合を取って定義した統合コードモデルを利用することで，対応する 7 種類の言語で記述された任意のソースコードを統合コードモデル上で表現可能である．

一方，ソースコードから統合コードオブジェクトへの変換とは逆に，統合コードオブジェクトをソースコードに変換する場合，必ずしも任意の統合コードオブジェクトをそれぞれ 7 種類の言語で記述されたソースコードで表現できるわけではない．例えば，Java 言語のソースコードから統合コードオブジェクトを生成して，そこで得られたクラス定義を表す統合コードオブジェクトから C 言語のソースコードを生成する場合，C 言語にはクラスという概念が存在しないため，1 対 1 の対応付けとして生成処理を実装できない．このような場合の対応は，対応言語拡張者に委ねられている．我々が開発した 7 種類の言語対応につい

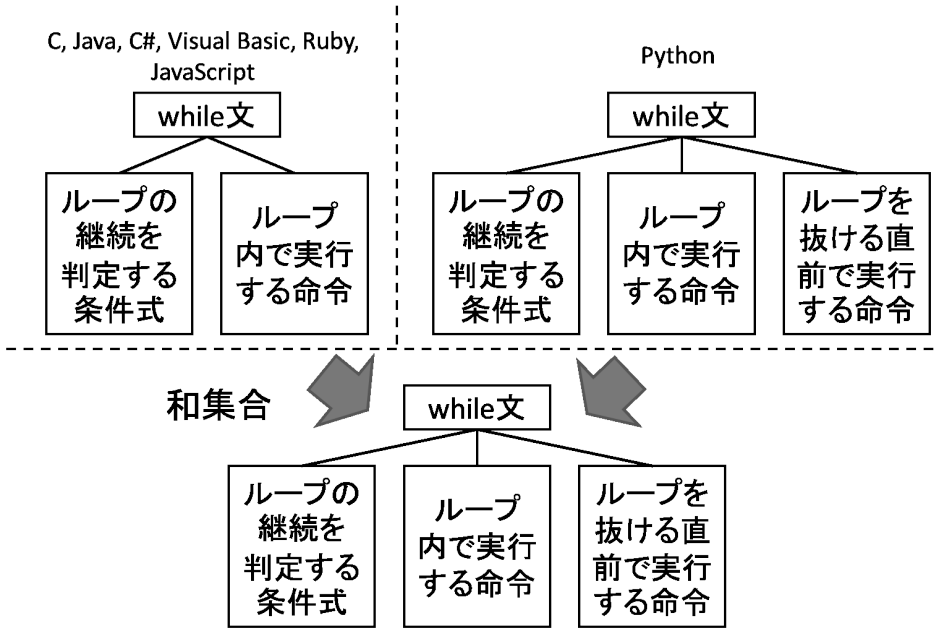


図 7　while 文の統合コードモデルの定義
Fig. 7　A unified code model of the while statement

```
1  Expression = If(Expression condition, Block body, Block elseBody)
2      | While(Expression condition, Block body, Block elseBody)
3      | DoWhile(Expression condition, Block body)
4      | For(Expression initializer, Expression condition, Expression step,
5          Block body, Block elseBody)
6      | FunctionDefinition(ModifierCollection modifiers, Type returnType,
7          Identifier name, ParameterCollection parameters, Block body)
```

図 8　ASDL を用いて記述した統合コードモデルの定義の一部
Fig. 8　The part of unified code model in ASDL

ては，例外を投げて終了する．しかし，C 言語は構造体や関数ポインタを組み合わせることでクラスを表現可能である．したがって，対応言語拡張者が，クラス定義の統合コードオブジェクトに対して，構造体や関数ポインタを組み合わせて表現したクラス定義のソースコードを生成する OC マッパーを記述すれば表現できる．たとえ該当する構文がなくとも，ソースコードを生成可能である．

4.2　対応言語拡張者向け API

UNICOEN は，対応言語の拡張コストを低減するために，対応言語拡張者向け API として，ANTLR[17] で構文解析した結果を.NETFramework 上の XML ツリーのオブジェクトとして表現する機能，XML ツリーのオブジェクト上を走査する機能を提供する．対応言語拡張者は，対応言語拡張者向け API を利用して，OC マッパーを構成する構文解析部，オブジェクト生成部，コード生成部の 3 つの機能部を実装する．ただし，構文解析部とオブジェクト生成部を 1 つの機能部として実装する場合や，コード生成部を持たずに片方向の変換機能のみを提供する場合もある．OC マッパーはツール開発者向け API が規定するインタフェースを持っており，インタフェースに従って実装しなければならない．OC マッパーは，ソースコードから統合コードオブジェクトを生成する処理，および，統合コードオブジェクトからソースコードを生成する処理を提供する．OC マッパーの利用例として，ソースコードと統合コードオブジェクト間の相互変換の様子とコードを図 9 と図 10 で示す．

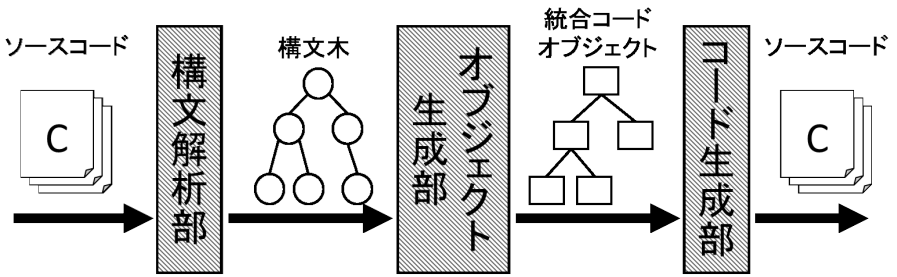


図 9　ソースコードと統合コードオブジェクトの相互変換
Fig. 9　A process of conversion and reverse conversion between source code and unified code objects

OC マッパーの実装方法は対応言語拡張者に委ねられているが，UNICOEN は ANTLR や既存のパーサーライブラリを利用して構文解析部を実装することを支援する．ANTLR は実績のあるコンパイラコンパイラであり，文法ファイルを入力することで LL(*) パーサーを自動生成する．ANTLR の Web サイトではいくつかの言語の文法ファイルが公開されており，その上，自由に文法ファイルをアップロードできるリポジトリページがある[18]．

```
var filePath = "code.java";
var ext = Path.GetExtension(filePath);
var progGen = UnifiedGenerators.GetProgramGeneratorByExtension(ext);
var uco = progGen.GenerateFromFile(filePath);
// 必要に応じて統合コードオブジェクトに対する解析や変形処理を記述
var code = progGen.CodeGenerator.Generate(uco);
```

図 10　UNICOEN を利用してソースコードと統合コードオブジェクトを相互変換する C#コード
Fig. 10　A code to inter-convert between source code and unified code objects in C# with UNICOEN

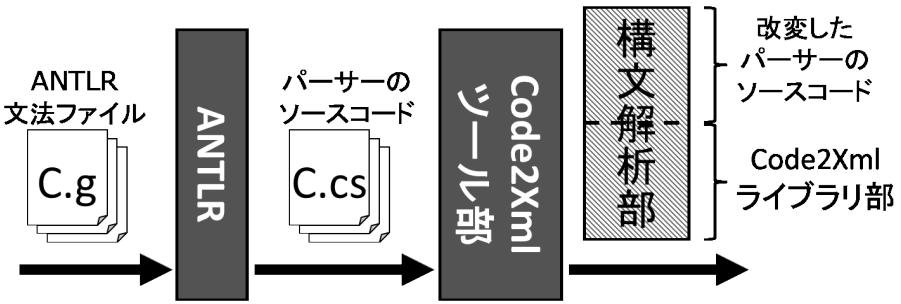


図 11　Code2Xml による構文解析部の自動生成
Fig. 11　A automatic generation of syntax analyzers with Code2Xml

我々は，ANTLR を用いて構文解析部を容易に実装できるように，UNICOEN のサブプロジェクトとして開発した Code2Xml を提供する[19)20)]．Code2Xml はツール部とライブラリ部から構成されている．ツール部は ANTLR が自動生成したパーサーのソースコードを自動修正する．そして，自動修正したソースコードとライブラリ部をリンクしてコンパイルすることで，対象となる言語のソースコードを構文解析して，構文木を.NET Framework 上の XML オブジェクトとして出力する機能を実現する．これによって，図 11 のように ANTLR 用の文法ファイルを用意するだけで，構文解析部を自動生成できる．

オブジェクト生成部では構文解析の結果から，統合コードオブジェクトを生成する．オブジェクト生成部では，XML オブジェクトで表現される構文木を解析して，統合コードモデルへのマッピング処理を行い，統合コードオブジェクトを生成する．その際，XML ノードの走査を容易に記述するために，いくつかのヘルパメソッドを共通処理として提供する．例えば，ある XML ノードを起点として，祖先ノード，子孫ノード，弟ノード，兄ノードなどを列挙する基本的な機能や，二項式の構文木が言語によらず類似することに着目して，演算子の意味や結合法則を与えることで式を解析する機能を提供する．なお，構文解析部とオブジェクト生成部は厳密に区別されておらず，両者をまとめて実装することもできるが，ANTLR を利用する場合は，明確に構文解析部とオブジェクト生成部が分かれる．ANTLR を利用しない場合であっても，既存のパーサーライブラリなどを利用して構文解析部で構文木を XML で出力することで，UNICOEN が提供する XML ツリーオブジェクト上を走査する機能を利用して，統合コードモデルへのマッピング処理を記述できる．我々が実際に開発した対応言語については 5.1 節で後述する．

以下に対応言語を拡張する手順を述べる．1. 拡張対象の言語仕様を調査して，ソースコードの構造化に必要なモデルを考える．2. 既存の統合コードモデルと考案したモデルを比較して差異を調べる．3. 得られた差異に対して，以下の手順 a,b のいずれかもしくは両方で統合コードモデルを拡張する．3.a. 対応する概念が統合コードモデルに存在しない場合は，それを表現するためのクラスを定義する．例えば，アスペクトという概念を統合コードモデルに追加する場合は，アスペクトに該当するクラスを定義する．3.b. 既に統合コードモデル上に対応する概念が存在していても既存のクラスで表現できない場合は，そのクラスにプロパティを追加する．例えば，Python の while 文における else 節のような概念が for 文にも存在する場合は，for 文を表現するクラスに else 節のプロパティを追加する．4. 対象の言語に対して拡張した統合コードモデル上でマッピングする OC マッパーを実装する．

クラスやプロパティの追加を行なっても，既存の OC マッパーやツールは影響を受けない．なぜなら，追加したクラスのインスタンスは既存の OC マッパーからは生成されないが，たまたまソースコード中に該当する構文が現れなかったのか，そもそも言語がそのような構文をサポートしていないのか区別できないためである．同様に，追加したプロパティは既存の OC マッパーでは null に初期化され，その統合コードオブジェクトが該当する要素を持たなかったのか，言語仕様上でそのような要素を持つことがないのか区別がつかない．

4.3　ツール開発者向け API

UNICOEN は，複数の言語に対応したツールの開発コストを低減するために，ツール開発者向け API として，ソースコードと統合コードオブジェクトを相互変換する機能と，統合コードモデル上で要素を抽出・追加・変更・削除する機能を提供する．これらの機能は，対応言語拡張者が各言語について OC マッパーを実装することで実現される．

統合コードオブジェクトの走査や変形の機能は，LINQ to XML と同じようなインタ

フェースの API を提供しており，.NET Framework が提供する LINQ と非常に親和性の高い操作系を提供する．例えば，図 12 と図 13 のように，LINQ to XML を用いて if という名前の要素ノード数を表示する C#言語で記述されたコードと，UNICOEN のツール開発者向け API を用いて if 文に該当する統合コードオブジェクト数を表示するコードは非常に似通っている．図 12 と図 13 の変数 ifs の型は，それぞれ IEnumerable⟨XElement⟩ と IEnumerable⟨IUnifiedElement⟩ である．XElement は XML の要素ノードを表し，IUnifiedElement は統合コードオブジェクトの要素を表す．どちらも集合を表現するために IEnumerable を利用しており，LINQ が提供する拡張メソッド Count() を利用できる．

```
1  var xml = XDocument.Load("code.xml");
2  var ifs = xml.Descendants("if");
3  Console.WriteLine("The number of if: " + ifs.Count());
```

図 12　if という名前の要素ノード数を表示する C#コード
Fig. 12　A code to enumerate elements with 'if' name in C#

```
1  var uco = UnifiedGenerators.GenerateProgramFromFile("code.java");
2  var ifs = uco.Descendants<UnifiedIf>();
3  Console.WriteLine("The number of if: " + ifs.Count());
```

図 13　UNICOEN を利用して if 文に該当する統合コードオブジェクト数を表示する C#コード
Fig. 13　A code to enumerate unified code objects of 'if' statement in C# with UNICOEN

図 14 で統合コードモデルとツール開発者向け API のクラス図を示す．ツール開発者向け API は，統合コードモデルを構成するクラス，統合コードモデルを構成するクラスのための拡張メソッドを持つクラス，統合コードオブジェクトとソースコードを相互変換するクラスから構成される．例えば，統合コードモデルとして，統合コードモデル上の要素であることを示す IUnifiedElement インタフェース，その具象クラス UnifiedElement，式を表現する IUnifiedExpression インタフェース，プログラム全体を示す UnifiedProgram クラスなどが挙げられる．これらのクラスやインタフェースは統合コードオブジェクトを解析したり変形するためにメソッドやプロパティを提供する．さらに，C#言語の拡張メソッドを用いて，統合コードオブジェクトの集合を表す既存の IEnumerable⟨IUnifiedElement⟩ インタフェースにメソッドを追加する．また，OC マッパーが継承して実装すべきソースコードから統合コードオブジェクトを生成する UnifiedProgramGenerator 抽象クラスと，統合コードオブジェクトからソースコードを生成する UnifiedCodeGenerator 抽象クラスが存在する．

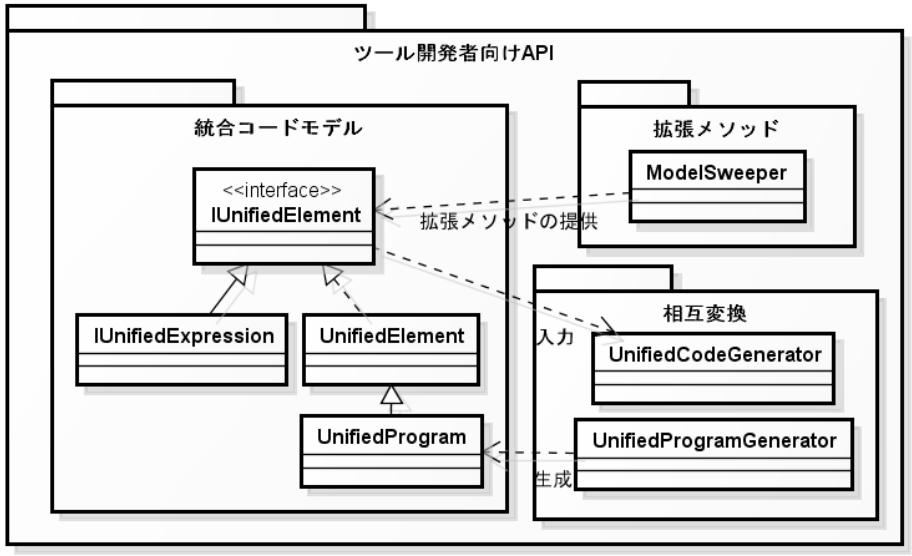


図 14　統合コードモデルとツール開発者向け API のクラス図
Fig. 14　A class diagram of the unified code model and the API for tool developers

ツール開発者が，ソースコード解析ツールを開発する場合は，ソースコードから統合コードオブジェクトを生成した上で，統合コードオブジェクトに対する解析処理を記述する．ソースコード変形ツールを開発する場合は，ソースコードから統合コードオブジェクトを生成して，さらに，統合コードオブジェクトを変形して，その上で，統合コードオブジェクトからソースコードを再生成する処理を記述する．

ソースコードのステートメント数を測定するツールの処理を図 15 で示す．統合コードオブジェクト上でステートメント数を測定する場合，ブロックの直下の子要素がステートメントに該当する統合コードオブジェクトなので，ブロックに該当する要素を全て抽出して，その直下の子要素の数を数えれば良い．図 16 でステートメント数を測定するコードを示す．

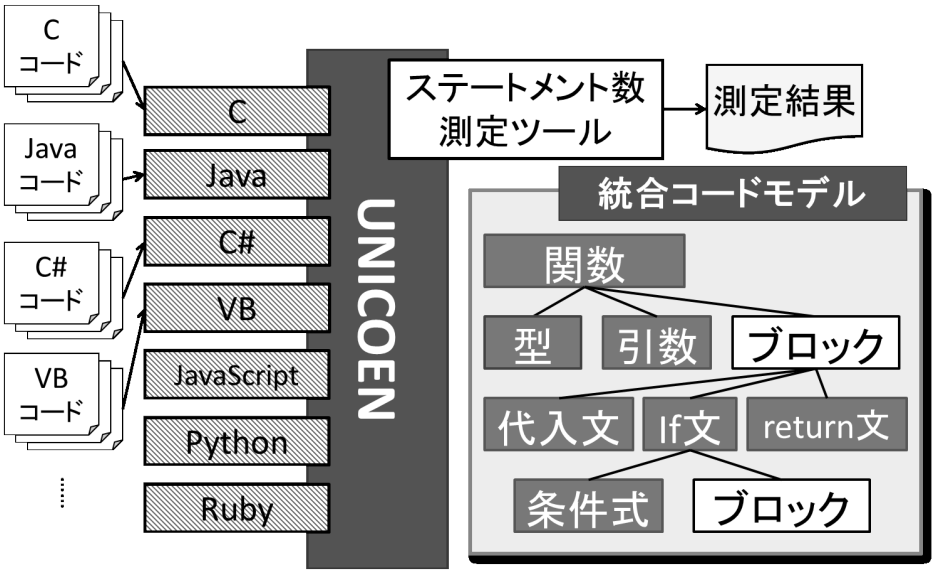


図 15　ソースコードのステートメント数を測定するツールの処理
Fig. 15　A processing of measurement tool for the number of statements

```
1  var uco = UnifiedGenerators.GenerateProgramFromFile("code.java");
2  var count = uco.Descendants<UnifiedBlock>().Sum(e => e.Count);
3  Console.WriteLine("The number of statements: " + count);
```

図 16　UNICOEN を利用してステートメント数を測定する C#言語コード
Fig. 16　A code to count statements in C# with UNICOEN

XML を解析するために XML を文字列として処理するよりも DOM ツリーとして処理する方が容易であるように，直接ソースコードを解析するよりも構造化されたオブジェクトを解析する方が容易である．例えば，UNICOEN が提供するツール開発者向け API の統合コードオブジェクトを走査する機能を利用することで，ある構文の数を数えるために，統合コードオブジェクトの中から特定の要素のみを抽出するといった処理が容易に記述できる．また，メソッド定義を追加したり，任意の演算子を変更したり，任意のステートメントを削除するために，統合コードオブジェクトが持つプロパティをの値を書き換えたり，Add や

表 2　実装した OC マッパーと既存の言語処理系のステートメント数の比較
その他処理系：GCC, GCJ, Mono, Rhino, IronPython, IronRuby
Table 2　A comparison of the number of statements between OC mappers and existing programming language processors: GCC, GCJ, Mono, Rhino, IronPython, IronRuby

言語	C	Java	C#	JavaScript	Python	Ruby
OC マッパー	727	1,003	399	626	636	501
既存の言語処理系	14,949	12,782	36,988	38,277	15,411	14,353

Remove メソッドを利用することで，得られた統合コードオブジェクトに対して子要素を追加，変更，削除する処理が容易に記述できる．このように，ツール開発者向け API を利用して構造化したオブジェクト上で解析処理や変形処理を実装することで，複数の言語に対応したソースコードを処理するツールの開発コストを大幅に削減する．

5. 評　　価

5.1　対応言語の実装

我々は C，Java，C#，Visual Basic，JavaScript，Python，Ruby の 7 種類の言語の OC マッパーを実装した．実装した OC マッパーと既存の言語処理系のステートメント数の比較を表 2 に示す．なお，既存の言語処理系として GCC の C コンパイラ，GCC の Java コンパイラ GCJ，Mono の C#コンパイラ MCS，Java VM 上で動作する JavaScript 処理系の Rhino，.NET Framework 上で動作する Python 処理系の IronPython と Ruby 処理系の IronRuby のコンパイラ部分を比較対象とした．なお，Rhino 以外の処理系はフレームワークの共通処理を除いて，ソースコードをマシン語もしくは中間言語に変換する処理に該当するソースコードのみを対象とするが，Rhino はフレームワークを利用していない上，処理系からコンパイラ部分を切り離せないため，処理系全体のステートメント数を提示する．

C，Java，JavaScript 言語について ANTLR と Code2Xml を用いて OC マッパーの構文解析部を実装した．これら 3 種類の言語の文法ファイルが ANTLR のリポジトリページで公開されており，ANTLR に入力してパーサーのプログラムを自動生成した．さらに，Code2Xml を利用して，生成したプログラムを.NET Framework 上の XML オブジェクトを生成するプログラムとなるように自動修正した．また，各言語のオブジェクト生成部とコード生成部を人手で実装した．オブジェクト生成部のステートメント数はそれぞれ 727, 1003，626 ステートメントであった．オブジェクト生成部は XML オブジェクトで表現される構文木から統合コードオブジェクトを，コード生成部は統合コードオブジェクトからソー

スコードを生成する．なお，構文木から統合コードオブジェクトを生成する機能の一部は，UNICOEN が提供する汎用的な共通処理を利用している．一方，GCC，GCJ，Rhino のステートメント数はそれぞれ 14949，12782，38277 ステートメントであった．

C#と Visual Basic 言語はオープンソースソフトウェアの NRefactory[21] を用いて構文解析部を実装した．NRefactory はこれらの言語のソースコードから抽象構文木を生成するパーサーライブラリである．オブジェクト生成部とコード生成部において，NRefactory が生成した抽象構文木と統合コードオブジェクトを相互変換する処理を人手で実装した．オブジェクト生成部のステートメント数は 616 ステートメントであった．一方，MCS のステートメント数は 36988 ステートメントであった．NRefactory が C#と Visual Basic 言語を区別せずに解析するため，両言語のオブジェクト生成部は共通である．また，ソースコードが公開された既存の Visual Basic 処理系がなかったため，表 2 には記載していない．

Python 言語は Python の標準ライブラリを用いて構文解析部を実装した．ソースコードから抽象構文木を生成して，それを XML ドキュメントで出力する Python スクリプトを記述した．UNICOEN は標準入出力を介して受け取った XML を.NET Framework 上の XML オブジェクトに変換する．他の言語と同様にオブジェクト生成部とコード生成部を人手で実装した．オブジェクト生成部のステートメント数は 636 ステートメントであった．一方，IronPython のコンパイラ部分のステートメント数は 15411 ステートメントであった．

Ruby 言語は.NET Framework で動作する Ruby 処理系の IronRuby[22] と Ruby 言語で記述されたパーサーライブラリである ruby_parser[23] を用いて構文解析部を実装した．ソースコードから抽象構文木を生成して，.NET Framework 上の XML オブジェクトに変換する IronRuby スクリプトを記述した．同様にオブジェクト生成部とコード生成部を人手で実装した．オブジェクト生成部のステートメント数は 501 ステートメントであった．一方，IronPython のコンパイラ部分のステートメント数は 14353 ステートメントであった．

C，Java，JavaScript 言語については ANTLR を利用したが，それ以外の 4 種類の言語については既存のパーサーライブラリを利用している．このように，UNICOEN では，構文解析部をコンパイラコンパイラやパーサーライブラリなど様々な既存ソフトウェアを利用して OC マッパーを実装できる．UNICOEN は完全な意味解析を行わずに統合コードモデルへのマッピング処理を行い，さらに，汎用的な共通処理として言語拡張社向け API を提供することで，既存の言語処理系と比較して実装すべきコード量を 20 倍から 50 倍程度も削減した．以上から，UNICOEN が問題 1 を解決することを確認した．

5.2　ツールの実装

5.2.1　メトリクス測定ツール UniMetrics

我々は McCabe の複雑度を測定するメトリクス測定ツール UniMetrics を Web アプリケーションとして UNICOEN を用いて開発した．McCabe の複雑度とは，サイクロマチック数，すなわち，「プログラム中の条件分岐数 + 1」を複雑度とするメトリクスである．McCabe の複雑度を測定できる既存ツールとして，Java 言語向けの Sonar[24] や Ruby 言語向けの Saikuro[25] が挙げられる．しかし，これらの測定ツールの測定基準には相違点が存在する．例えば，Sonar は拡張 for 文を条件分岐としてカウントするが，Saikuro は拡張 for 文に該当する each メソッドを条件分岐としてカウントしない．その上，これらの測定ツールの測定結果を 1 つにまとめて表示するツールは存在しない．したがって，Java と Ruby 言語を用いて開発したソフトウェア全体の複雑度を測定して評価することは困難である．

そこで，我々は上述した問題を解決するために UNICOEN 上で UniMetrics を開発した．UniMetrics は，同じ測定基準を用いて複数の言語における McCabe の複雑度を測定でき，測定結果を 1 つのグラフにまとめて表示する．UniMetrics は GitHub 上のプロジェクトを対象として，McCabe の複雑度を測定できる．GitHub 上のプロジェクトの URL を入力として与えると，ソースコードをダウンロードして，McCabe の複雑度を測定して，測定結果を棒グラフにして表示する．利用者は UniMetrics をダウンロードして実行環境を整える必要がない．以上から，UNICOEN が言語間のツールの差異を低減して，問題 2 を解決することを確認した．UniMetrics の実行結果を図 17 に示す．

McCabe の複雑度を測定するプログラムのステートメント数について，既存の測定ツール Saikuro と UNICOEN 上で開発した UniMetrics の比較結果を表 3 に示す．Saikuro は複雑度を測定するプログラムのステートメント数が 321 ステートメントであるのに対して，UniMetrics ではわずか 1 ステートメントで記述される．その上，Saikuro は Ruby 言語のソースコードのみを測定対象とするが，UniMetrics では UNICOEN が対応する全ての言語のソースコードを測定対象とする．我々は 7 種類の言語の OC マッパーを実装したため，UniMetrics は 7 種類の言語に対応している．今後，新たな言語の OC マッパーを実装すれば，UniMetrics のソースコードに一切の変更を加えずに，UniMetrics が対応する言語を拡張できる．以上から，UNICOEN がツールの開発コストを低減して，問題 1 を解決することを確認した．なお，UNICOEN を利用して記述した McCabe の複雑度を測定するコードは図 18 のとおりである．

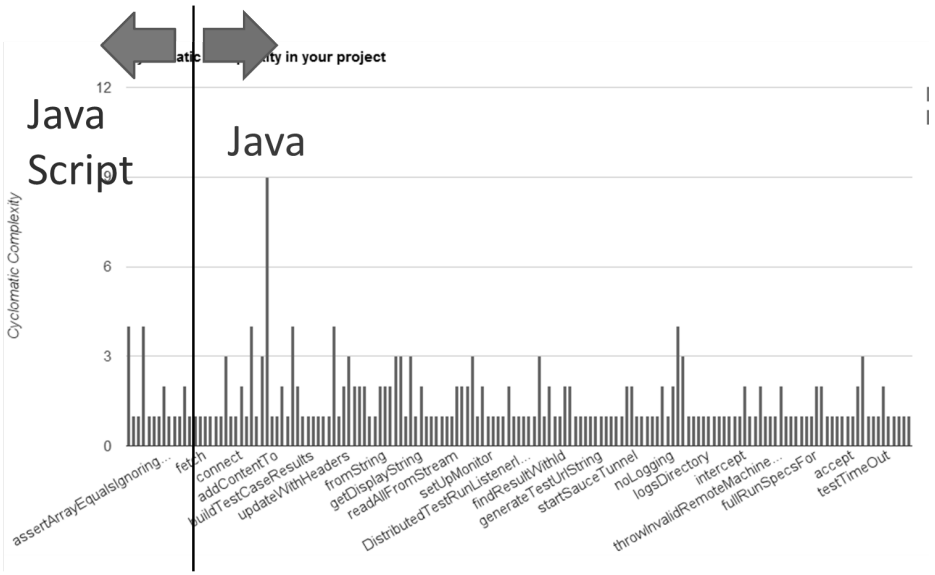


図 17　UNICOEN を利用して開発した McCabe の複雑度の測定ツール UniMetrics の実行画面
Fig. 17　A display of UniMetrics which is developed with UNICOEN to measure McCabe complexity

表 3　既存ツール Saikuro と UNICOEN を利用する UniMetrics における対応言語と測定処理のステートメント数による比較
Table 3　A comparisons of the supported languages and the numebr of statements between Saikuro and UniMetrics

	対応言語	測定処理のステートメント数
Saikuro	Ruby	321
UniMetrics	C, Java, C#, Visual Basic, JavaScript, Python, Ruby	1

5.2.2　CodeCity の対応言語の拡張ツール UniCodeWorld

CodeCity[26] は，メトリクスの測定結果を読み込んで，測定結果を街に見立てて可視化するソフトウェア品質の可視化ツールである．街全体が 1 つのソフトウェアを表しており，建物 1 つ 1 つがソフトウェアを構成するモジュールを表す．そして，建物の特徴を任意のメトリクスの測定結果と対応付けられる．例えば，建物 1 つをクラス，高さをクラスのコード行数，胴回りの太さをクラスの持つメソッドの数として表現できる．CodeCity は専用のフォーマットでメトリクスの測定結果を入力する必要がある．Java，C++，C#言語に対応した既存ツールがそれぞれ存在しており，1 つの言語で記述されたソースコードのみを解析できる．我々はより多くの言語に対応した CodeCity 専用のメトリクス測定ツール UniCodeWorld を UNICOEN を利用して開発した．Java 言語のみで開発された jEdit について，既存ツールで得た測定値を CodeCity で可視化した結果が図 19 である．Java と JavaScript 言語で開発された JsUnit について，UniCodeWorld で得た測定値を CodeCity で可視化した結果が図 20 である．

```
1  var count = UnifiedGenerators.GenerateProgramFromFile("code.java")
2    .Descendants<UnifiedIf, UnifiedFor, UnifiedForeach, UnifiedWhile,
3      UnifiedDoWhile, UnifiedCase>().Count() + 1;
```

図 18　UNICOEN を利用して記述した McCabe の複雑度を測定する C#コード
Fig. 18　A code to measure McCabe complexity in C# with UNICOEN

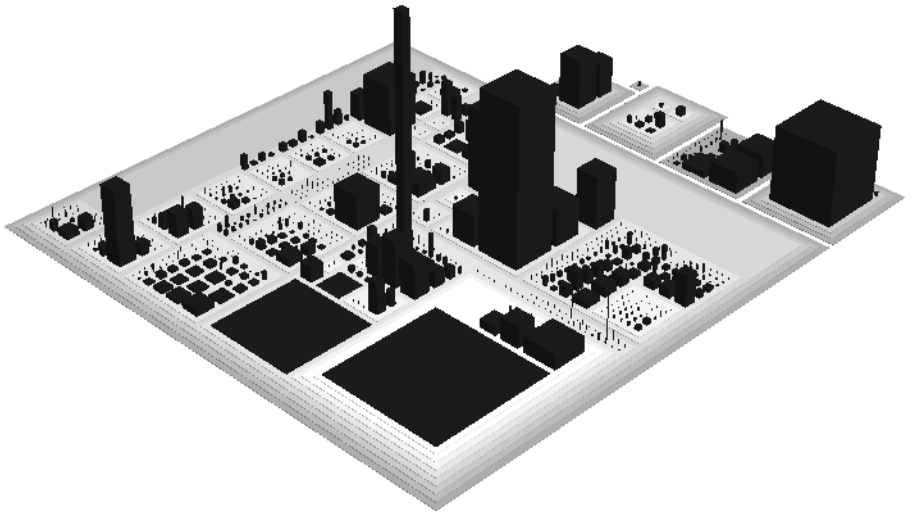


図 19　CodeCity による Java 言語のみで開発された jEdit の品質の可視化
Fig. 19　A visualization of jEdit which is written in Java with CodeCity

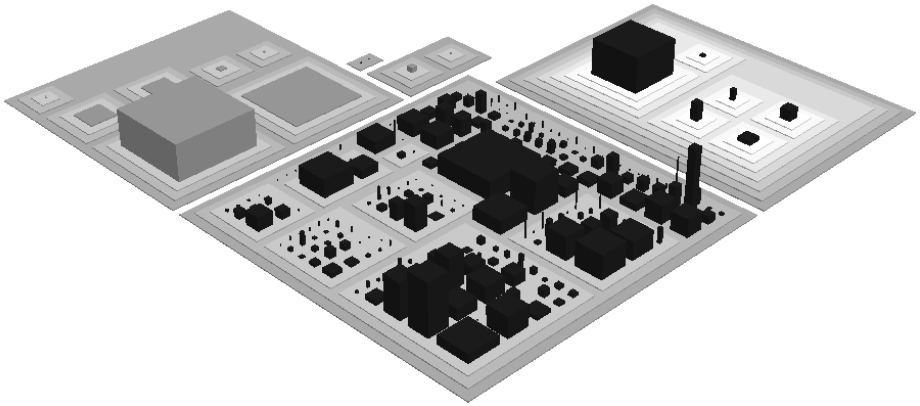


図 20　UniCodeWorld と CodeCity による Java と JavaScript 言語で開発された JsUnit の品質の可視化
Fig. 20　A visualization of JsUnit which is written in Java and JavaScript with UniCodeWorld and CodeCity

図 19 は建物の高さをメソッド数，胴回りの太さをフィールド数に対応付けて可視化した．一方，図 20 は，Java と JavaScript 言語で記述したクラスをそれぞれ黒色と灰色に，建物の高さをステートメント数に，胴回りの太さをメソッド数に対応付けて可視化した．図 20 のように，UniCodeWorld によって建物の色付けを変更して 1 つの街として表示できる．既存ツールは言語毎に開発されているため，図 19 のように 1 つの言語で開発されたソフトウェアのみを可視化できる．一方，UniCodeWorld を利用することでは，図 20 のように複数の言語で開発されたソフトウェアも可視化できる．これにより，可視化の対象となるソフトウェアがどのような言語で開発されたのか，その割合やモジュール化の様子を一目で分かるようになった．以上から，UNICOEN が言語間のツールの差異を低減して，問題 2 を解決することを確認した．

図 20 の JavaScript 言語で構成される街の一部に，1 つだけ突出した建物が見られる．これは，粒度の小さいクラスによってモジュール化せずに 1 つのクラスにコードを詰め込んでしまうという，JavaScript 言語で記述されるプログラムの傾向を表す．このように，UniCodeWorld と CodeCity を用いて可視化した結果に，言語の特色が現れることが判明した．UNICOEN がソフトウェア品質の可視化ツールの新たな可能性を開いたことを示唆する．

表 4　既存ツール PMCS と UNICOEN を利用する UniCodeWorld における対応言語とステートメント数の比較
Table 4　A comparisons of the supported languages and the numebr of statements between PMCS and UniCodeWorld

	対応言語	測定処理のステートメント数
PMCS	C#	1478
UniCodeWorld	C，Java，C#，Visual Basic，JavaScript，Python，Ruby	203

CodeCity による可視化のための専用ツールのステートメント数について，既存ツール PMCS[27] と UniCodeWorld の比較結果を表 4 に示す．PMCS では，C#言語のみに対応してプログラムのステートメント数が 1478 ステートメントである．一方，UniCodeWorld は，UNICOEN が対応する全ての言語に対応してわずか 203 ステートメントで記述される．Java，C++，C#言語以外の言語では CodeCity を利用できなかったが，UNICOEN によって，新たに Visual Basic，Ruby，Python，JavaScript 言語に対応した．UNICOEN が言語の種類によりツールの恩恵を受けられなかった状況を打破した．以上から，UNICOEN が開発コストを低減して，問題 1 を解決することを確認した．

5.2.3　アスペクト指向プログラミング処理系 UniAspect

我々は複数言語に対応する AOP 処理系 UniAspect を UNICOEN を用いて開発した．例えば，既存の AOP 処理系として Java 言語向けに AspectJ や JavaScript 言語向けに AOJS が存在している．しかし，これらの AOP 処理系は特定の言語に特化して実装されており，アスペクトの文法と記述能力が異なる．そのため，複数の言語に対応した AOP 処理系が存在せず，複数言語を使用するソフトウェア開発で AOP を導入する場合，利用者は言語毎に AOP 処理系のアスペクト記述方法を学んだ上で，別々にアスペクトを記述する必要がある．

そこで，上述した問題を解決するために UNICOEN 上で UniAspect を開発した．UniAspect のアスペクトは織り込むコードのアドバイスと織り込み先を示すポイントカットから構成される．ポイントカットは言語非依存に記述でき，アドバイスは織り込み先の言語毎に記述する．UniAspect は 1 つのアスペクトを複数の言語のソースコードに適用できるため，利用者の学習コストを低減して，より良いモジュール化を実現する．図 4 と図 3 のアスペクトについて，UniAspect を用いて 1 つのアスペクトに記述した結果が図 21 である．

アスペクトの適用例として，UniAspect による統合コードオブジェクト上で，関数の実行部分の先頭にコードを挿入する様子を図 22 で示す．

図 21 のアスペクトが織り込まれる処理の流れは以下のとおりである．1. プログラム全体に該当する統合コードオブジェクトから関数に該当する要素を抽出する．2. 得られた関数

```
aspect Logger {
  pointcut allMethod() : execution(* *.*());

  before : allMethod() {
    @Java {
      System.out.println(JOINPOINT_NAME + " is executed.");
    }end
    @JavaScript {
      console.log(JOINPOINT_NAME + " is executed.");
    }end
  }
}
```

図 21　メソッドの実行に対するロギングする UniAspect のアスペクト
Fig. 21　A logging code for executing methods in UniAspect

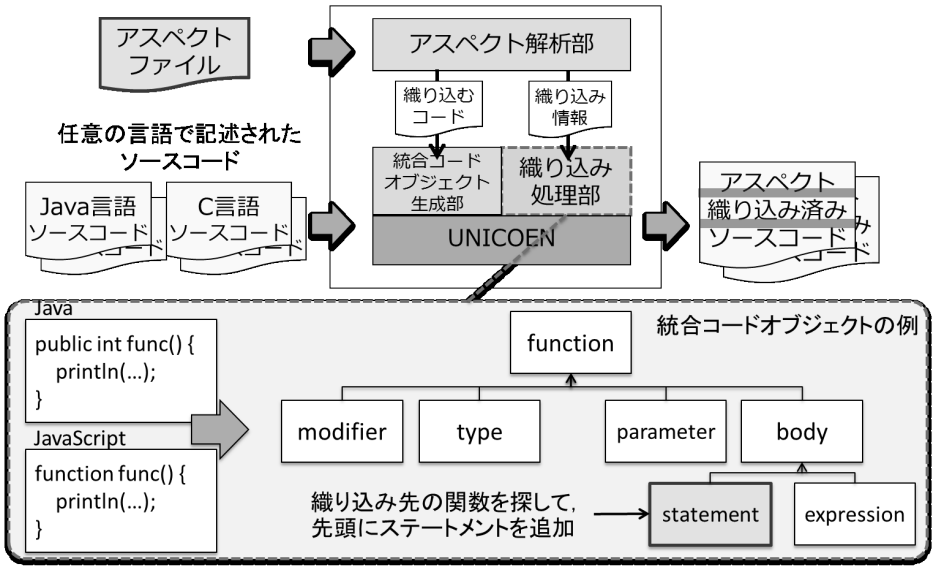


図 22　UniAspect による統合コードオブジェクト上でのアスペクトの織り込み
Fig. 22　Weaving aspect on unified code model using UniAspect

の中から，関数名や戻り値の型を参照して，ユーザが指定した条件に当てはまる関数に絞り込む．3. 絞り込んだ関数が持つブロックの先頭に，指定したコードに該当する統合コードオブジェクトを挿入する．このようなアスペクトの織り込み処理は，UNICOEN が提供するツール作成者向け API を用いて言語非依存に実装可能である．なお，動的型付け言語では戻り値の型が明示的に示されないため，一部の言語において戻り値の型の条件は無視される．以上から，UNICOEN が複数言語に対応する統一的なツールの開発を支援して，問題 1 を解決することを確認した．

6. 関 連 研 究

(1)　Lattner ら[28] はコンパイラフレームワーク LLVM を提案した．同様のフレームワークとして GCC が，関連技術として Java VM や.NET Framework など中間言語の実行環境が挙げられる．これらの既存ソフトウェアは，各言語のソースコードのセマンティクスを完全に解析して中間言語に変換する．中間言語は UNICOEN の統合コードモデルに該当するため，中間言語を介することで言語を意識せずにツールを開発できる[29]．しかし，既存ソフトウェアはコンパイラや言語処理系など，セマンティクスを完全に解釈した上でソースコードを処理するツールを対象とする．そのため，これらの中間言語にマッピングする際，完全に意味解析を行う必要があり対応する言語を追加するコストが非常に大きい．一方，UNICOEN は意味解析を完全に行わないことで，対応する言語の追加およびツールの開発コストを削減する．実際，これらの既存ソフトウェアは LLVM を除いて 10 年以上開発されているにもかかわらず，我々が半年程度で UNICOEN に対応させた言語全てに対応するものは存在しない．様々な言語のセマンティクスを失わずにマッピングできるような共通の中間言語は，マシン語に近い非常に抽象度の低い仕様となる．そのため，既存ソフトウェアを利用してツールを開発する際，厳密なセマンティクス情報を得られるものの，ソースコードの抽象度は失われており，シンタックス情報は得られない．このことは，既存ソフトウェアでは，コード整形などシンタックス情報を必要とするツール開発が困難であることを示す．一方，UNICOEN は構文解析の結果に基づいてソースコードを構造化するため，シンタックスの情報に基づく処理を実装しやすい．そのため，コード整形ツールなどは UNICOEN を利用することで容易に実装できる．このようなシンタックスに着目するツールの開発を支援するソフトウェアは存在しておらず，この相違点こそが UNICOEN が革新的なソフトウェアである所以である．

(2)　Higo ら[30] は複数のプログラミング言語に対応したメトリクス測定フレームワーク MASU を提案した．MASU はメトリクス測定の観点から必要なセマンティクスを解

釈して，言語非依存な抽象構文木を構築する．言語非依存な抽象構文木を走査することでメトリクスを測定でき，MASU のプラグインとして測定処理を実装することで，MASU を通して Java, C#, Visual Basic 言語のソースコードのメトリクスを測定できる．一方，UNICOEN は MASU が対応する言語全てに対応している上，オブジェクト指向プログラミング言語以外や動的型付け言語にも対応する．その上，ソースコードの解析処理のみならず，変形処理も言語非依存に記述可能である．

7. ま と め

本論文では，複数言語対応のソースコード処理フレームワーク UNICOEN を提案した．UNICOEN は，シンタックスに基づいてソースコードを統合コードモデル上のオブジェクトにマッピングして，汎用的な共通処理をツール開発者向け API および言語拡張者向け API として提供することで，言語とツールの多対多の関係を言語と UNICOEN 間およびツールと UNICOEN 間の多対一の関係に簡略化した．我々は，UNICOEN に C，Java，C#，Visual Basic，JavaScript，Python，Ruby 言語の対応を追加して，その上で，これらの言語に対応したメトリクス測定ツール，CodeCity の言語対応の拡張，AOP 処理系を開発した．そして，対応言語の拡張において既存の言語処理系と比較して大幅に少ない記述量で実装可能であること，さらに，ツール開発において言語非依存な形で，かつ既存ツールと比較して少ない記述量で実装可能であることを確認した．また，開発したツールが複数言語に対応した統一的な機能を提供して，複数存在するツール間の差異を低減することを確認した．以上から，UNICOEN が問題 1 および問題 2 を解決することを確認した．

今後の展望として，以下のような点が挙げられる．

(1) 関数型言語の対応の実装
現状で UNICOEN が対応する言語は手続き型言語のみとなっており，Haskell や OCaml のような関数型言語の対応を実装していない．そこで，UNICOEN にこれらの言語を対応させることで，統合コードモデルが手続き型言語のみならず関数型言語においても有用であることを示す必要がある．

(2) 対応言語拡張を支援するツールの開発
現状では，UNICOEN の対応言語を拡張する際，人手で OC マッパーを開発する必要がある．また，新たな言語を拡張する際に，場合によっては統合コードモデルを人手で拡張する必要がある．このような作業は既存の処理系において対応言語を拡張するよりも容易であると考えられるが，ソースコードから統合コードオブジェクトへのマッピング記述から OC マッパーを自動生成する処理系や，ASDL のような統合コードモデルの仕様記述から統合コードモデルに該当するクラスを自動生成する処理系の開発が考えられる．

謝辞　UNICOEN は IPA の 2010 年度未踏 IT 人材発掘・育成事業の支援を受けて開発しています．UNICOEN の一部はガイオ・テクノロジー株式会社の支援を受けて開発しています．UNICOEN の開発に協力頂いた岩澤 宏希氏に感謝します．

参 考 文 献

1) Breslav, A.: Welcome - Kotlin - Confluence, JetBrains (online), available from ⟨http://confluence.jetbrains.net/display/Kotlin/⟩
2) The Eclipse Foundation: Eclipse Xtend, The Eclipse Foundation (online), available from ⟨http://www.eclipse.org/xtend/⟩
3) Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P. and Vouk, M.A.: On the Value of Static Analysis for Fault Detection in Software, *IEEE Trans. Softw. Eng.*, Vol.32, pp.240–253 (online), DOI:10.1109/TSE.2006.38 (2006).
4) Hovemeyer, D. and Pugh, W.: Finding bugs is easy, *SIGPLAN Not.*, Vol.39, pp. 92–106 (online), DOI:http://doi.acm.org/10.1145/1052883.1052895 (2004).
5) Crockford, D.: JSLint,The JavaScript Code Quality Tool, , available from ⟨http://www.jslint.com/⟩ (accessed 2012-02-04).
6) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, London, UK, UK, Springer-Verlag, pp.327–353 (online), available from ⟨http://dl.acm.org/citation.cfm?id=646158.680006⟩ (2001).
7) Washizaki, H., Kubo, A., Mizumachi, T., Eguchi, K., Fukazawa, Y., Yoshioka, N., Kanuka, H., Kodaka, T., Sugimoto, N., Nagai, Y. and Yamamoto, R.: AOJS: aspect-oriented javascript programming framework for web development, *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '09, New York, NY, USA, ACM, pp. 31–36 (online), DOI:http://doi.acm.org/10.1145/1509276.1509285 (2009).
8) TIOBE Software BV: TIOBE Software: The Coding Standards Company, TIOBE Software BV (online), available from ⟨http://www.tiobe.com/index.php/content/paperinfo/tpci/⟩ (accessed 2012-02-04).
9) Johnson, S.C.: Lint, a C Program Checker, *COMP. SCI. TECH. REP*, pp.78–1273 (1978).

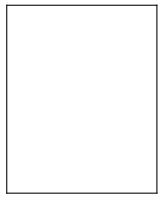
10) Logilab: pylint 0.25.1 : Python Package Index, , available from ⟨http://pypi.python.org/pypi/pylint⟩ (accessed 2012-02-04).
11) Pitrou, A.: Python2orPython3 - PythonInfo Wiki, Python Software Foundation (online), available from ⟨http://wiki.python.org/moin/Python2orPython3⟩ (accessed 2012-02-04).
12) Roubtsov, V.: EMMA: a free Java code coverage tool, , available from ⟨http://emma.sourceforge.net/⟩ (accessed 2012-02-04).
13) Batchelder, N.: coverage 3.5.1 : Python Package Index, , available from ⟨http://pypi.python.org/pypi/coverage⟩ (accessed 2012-02-04).
14) Sakamoto, K., Ohashi, A., Ota, D., Iwasawa, H. and Kamiya, T.: NuGet Gallery, The UNICOEN Project (online), available from ⟨http://www.unicoen.net/⟩ (accessed 2012-02-04).
15) Sakamoto, K., Ohashi, A., Ota, D., Iwasawa, H. and Kamiya, T.: UnicoenProject/UNICOEN - GitHub, The UNICOEN Project (online), available from ⟨https://github.com/UnicoenProject/UNICOEN⟩ (accessed 2012-02-04).
16) Sakamoto, K.: NuGet Gallery(UNICOEN 1.0.3.28), The UNICOEN Project (online), available from ⟨http://nuget.org/packages/UNICOEN⟩ (accessed 2012-02-04).
17) Parr, T.J. and Quong, R.W.: ANTLR: a predicated-LL(k) parser generator, *Softw. Pract. Exper.*, Vol.25, pp.789–810 (online), DOI:10.1002/spe.4380250705 (1995).
18) Parr, T.: ANTLR Parser Generator, , available from ⟨http://www.antlr.org/⟩ (accessed 2012-02-04).
19) Sakamoto, K.: code2xml, , available from ⟨http://code.google.com/p/code2xml/⟩ (accessed 2012-02-04).
20) Sakamoto, K.: NuGet Gallery(Code2Xml 1.2.0.47), , available from ⟨https://nuget.org/packages/Code2Xml⟩ (accessed 2012-02-04).
21) Krüger, M.: icsharpcode/NRefactory - GitHub, , available from ⟨https://github.com/icsharpcode/NRefactory⟩ (accessed 2012-02-04).
22) Schementi, J., de Icaza, M. and Matousek, T.: IronRuby, , available from ⟨http://ironruby.codeplex.com/⟩ (accessed 2012-02-04).
23) RyanDavis, E.H.: icsharpcode/NRefactory - GitHub, , available from ⟨http://rubyforge.org/projects/parsetree/⟩ (accessed 2012-02-04).
24) SonarSource: Sonar, , available from ⟨http://www.sonarsource.org/⟩ (accessed 2012-02-04).
25) Blut, Z.: Saikuro : A Cyclomatic Complexity Analyzer, , available from ⟨http://saikuro.rubyforge.org/⟩ (accessed 2012-02-04).
26) Wettel, R., Lanza, M. and Robbes, R.: Software systems as cities: a controlled experiment, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA, ACM, pp. 551–560 (online), DOI:http://doi.acm.org/10.1145/1985793.1985868 (2011).
27) Doernenburg, E.: erikdoe / PMCS / overview - Bitbucket, , available from ⟨https://bitbucket.org/erikdoe/pmcs/⟩ (accessed 2012-02-04).
28) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp.75– (online), available from ⟨http://dl.acm.org/citation.cfm?id=977395.977673⟩ (2004).
29) Lafferty, D. and Cahill, V.: Language-independent aspect-oriented programming, *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, New York, NY, USA, ACM, pp.1–12 (online), DOI:http://doi.acm.org/10.1145/949305.949307 (2003).
30) Higo, Y., Saitoh, A., Yamada, G., Miyake, T., Kusumoto, S. and Inoue, K.: A Pluggable Tool for Measuring Software Metrics from Source Code, *Proceedings of the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, IWSM-MENSURA '11, Washington, DC, USA, IEEE Computer Society, pp.3–12 (online), DOI:http://dx.doi.org/10.1109/IWSM-MENSURA.2011.43 (2011).

付　　録

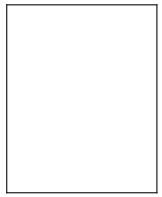
A.1　統合コードモデルの ASDL

統合コードモデルの ASDL の全体を表 23-26 にて示す．なお，ASDL の左辺と右辺で現れる要素は全て UNICOEN 上のクラスとして定義していて，右辺のクラスが左辺のクラスを継承する形で実装している．

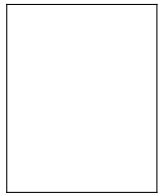
(平成 22 年 7 月 17 日受付)
(平成 22 年 9 月 17 日採録)



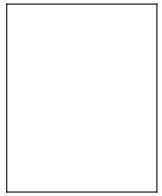
坂本　一憲（正会員）



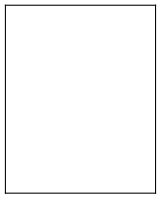
大橋　　昭（正会員）



太田　大地（正会員）



鷲崎　弘宜（正会員）



深澤　良彰（正会員）

```
IElement
 = Element
 | ElementCollection
 | IExpression

Element
 = Program(Block body)
 | Parameter(AnnotationCollection annotations, ModifierCollection modifiers, Type type,
   IdentifierCollection names, IExpression defaultValue, IExpression annotationExpression)
 | Modifier(string name)
 | GenericParameter(Type type, TypeConstrainCollection constrains,
   ModifierCollection modifiers)
 | GenericArgument(IExpression type, ModifierCollection modifiers,
   TypeConstrainCollection constrains)
 | Comment(string comment)
 | Case(IExpression condtion, Block body)
 | Argument(IExpression value, Identifier target, ModifierCollection modifiers)
 | Annotation(IExpression name, ArgumentCollection arguments)
 | TypeConstrain()
 | PropertyDefinitionPart(AnnotationCollection annotations, ModifierCollection modifiers,
   Block body)
 | VariableDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
   Type type, Identifier name, IExpression initialValue, ArgumentCollection arguments,
   IntegerLiteral bitField, Block body)
 | LinqQuery()
 | OrderByKey(IExpression expression, bool ascending)
 | BinaryOperator(string sign, BinaryOperatorKind kind)
 | UnaryOperator(string sign, UnaryOperatorKind kind)

TypeConstrain
 = ValueConstrain(Type type)
 | SuperConstrain(Type type)
 | ReferenceConstrain(Type type)
 | ImplementsConstrain(Type type)
 | ExtendConstrain(Type type)
 | EigenConstrain(Type type)
 | DefaultConstrain(Type type)

ElementCollection
 = AnnotationCollection(IList<Annotation> elements)
 | ArgumentCollection(IList<Argument> elements)
 | CaseCollection(IList<Case> elements)
 | CatchCollection(IList<Catch> elements)
 | ExpressionCollection(IList<Expression> elements)
 | GenericArgumentCollection(IList<GenericArgument> elements)
 | GenericParameterCollection(IList<GenericParameter> elements)
 | IdentifierCollection(IList<Identifier> elements)
 | ModifierCollection(IList<Modifier> elements)
 | OrderByKeyCollection(IList<OrderByKey> elements)
 | ParameterCollection(IList<Parameter> elements)
 | TypeCollection(IList<Type> elements)
 | TypeConstrainCollection(IList<TypeConstrain> elements)
```

図 23　ASDL を用いて記述した統合コードモデルの定義
Fig. 23　The unified code model in ASDL

```
IExpression
 = Call(IExpression target, ArgumentCollection args,
   GenericArgumentCollection genericArguments, Proc proc)
 | Cast(Type type, IExpression createExpression)
 | Indexer(IExpression current, ArgumentCollection create)
 | KeyValue(IExpression key, IExpression value)
 | Label(string name)
 | New(IExpression target, ArgumentCollection arguments,
   GenericArgumentCollection genericArguments, ArrayLiteral initialValues, Block body)
 | Property(string delimiter, IExpression owner, IExpression name)
 | Slice(IExpression initializer, IExpression condition, IExpression step)
 | Switch( IExpression value, CaseCollection cases)
 | Catch(TypeCollection types, IExpression assign, Block body,
   AnnotationCollection annotations, ModifierCollection modifiers)
 | If(IExpression condition, Block body, Block falseBody)
 | Lambda(Identifier name, ParameterCollection parameters, Block body)
 | Proc(ParameterCollection parameters, Block body)
 | Try(Block body, CatchCollection catches, Block elseBody, Block finallyBody)
 | ConstructorLike<TSelf>(Block body, AnnotationCollection annotations,
   ModifierCollection modifiers, ParameterCollection parameters,
   GenericParameterCollection genericParameters, TypeCollection throws)
 | DoWhile(IExpression condition, Block body, Block falseBody)
 | For(IExpression initializer, IExpression condition, IExpression step, Block body)
 | Foreach(IExpression element, IExpression set, Block body, Block elseBody)
 | While(IExpression condition, Block body, Block elseBody)
 | Fix(IExpression value, Block body)
 | Synchronized(IExpression value, Block body)
 | Using(ExpressionCollection expressions, Block body)
 | With(IExpression value, Block body)
 | ComprehensionBase()
 | MapComprehension(KeyValue element, ExpressionCollection generator)
 | ClassLikeDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
   IExpression name, GenericParameterCollection genericParameters,
   TypeConstrainCollection constrains, Block body)
 | EventDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
   Type type, Identifier name, ParameterCollection parameters,
   PropertyDefinitionPart adder, PropertyDefinitionPart remover)
 | FunctionDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
   Type type, GenericParameterCollection genericParameters, Identifier name,
   ParameterCollection parameters, TypeCollection throws, Block body,
   IExpression annotationExpression)
 | PropertyDefinition(AnnotationCollection annotations, ModifierCollection modifiers,
   Type type, Identifier name, ParameterCollection parameters,
   PropertyDefinitionPart getter, PropertyDefinitionPart setter)
 | Identifier(string name)
 | Break(IExpression value)
 | Continue(IExpression value)
 | Goto(Identifier value)
 | Redo()
 | Retry()
 | Return(IExpression value)
 | Throw(IExpression value, IExpression data, IExpression trace)
```

図 24　ASDL を用いて記述した統合コードモデルの定義（続き 1）
Fig. 24　The unified code model in ASDL(Cont.1)

```
  | YieldBreak(IExpression value)
  | YieldReturn(IExpression value)
  | LinqExpression()
  | ArrayLiteral()
  | IterableLiteral()
  | ListLiteral()
  | MapLiteral()
  | SetLiteral()
  | TupleLiteral()
  | Literal()
  | Range(IExpression min, IExpression max)
  | BinaryExpression(IExpression leftHandSide, BinaryOperator binaryOperator,
    IExpression rightHandSide)
  | TernaryExpression(IExpression condition, IExpression trueExpression,
    IExpression falseExpression)
  | UnaryExpression(IExpression operand, UnaryOperator unaryOperator)
  | Alias(IExpression value, IExpression alias )
  | Assert(IExpression value, IExpression message)
  | Default(Type type)
  | Defined(IExpression value)
  | Delete(IExpression value)
  | Exec(IExpression value)
  | Import(IExpression name, string alias, IExpression member, ModifierCollection modifiers)
  | Pass(IExpression value)
  | Print(IExpression value)
  | PrintChevron(IExpression value)
  | Sizeof(IExpression expression)
  | StringConversion(IExpression value)
  | Typeof(IExpression type)
  | Type(IExpression basicExpression)
  | Block(IList<IUnifiedExpress> elements)
  | VariableDefinitionList(IList<VariableDefinition> elements)

ClassLikeDefinition
 = AnnotationDefinition()
 | ClassDefinition()
 | EigenClassDefinition()
 | EnumDefinition()
 | InterfaceDefinition()
 | ModuleDefinition()
 | NamespaceDefinition()
 | StructDefinition()
 | UnionDefinition()

ConstructorLike
 = Constructor()
 | InstanceInitializer()
 | StaticInitializer()
```

図 25　ASDL を用いて記述した統合コードモデルの定義（続き 2）
Fig. 25　The unified code model in ASDL(Cont.2)

```
ComprehensionBase
 = IterableComprehension(IExpression element, ExpressionCollection generator)
 | ListComprehension(IExpression element, ExpressionCollection generator)
 | SetComprehension(IExpression element, ExpressionCollection generator)

Identifier
 = LabelIdentifier(string name)
 | SuperIdentifier(string name)
 | ThisIdentifier(string name)
 | TypeIdentifier(string name)
 | VariableIdentifier(string name)
 | VaueIdentifier(string name)

LinqQuery
 = FromQuery(VariableIdentifier receiver, IExpression source, Type receiverType)
 | GroupByQuery(IExpression element, IExpression key, VariableIdentifier receiver)
 | JoinQuery(VariableIdentifier receiver, IExpression joinSource,
   IExpression firstEqualsKey, IExpression secondEqualsKey)
 | LetQuery(VariableIdentifier variable, IExpression expression)
 | OrderByQuery(OrderByKeyCollection keys)
 | SelectQuery(IExpression expression, VariableIdentifier receiver)
 | WhereQuery(IExpression condition)

Literal
 = NullLiteral()
 | TypedLiteral()

TypedLiteral
 = IntegerLiteral(BigInteger value)
 | BooleanLiteral(bool value)
 | CharLiteral(string value)
 | FractionLiteral(double value, FractionLiteralKind kind)
 | RegularExpressionLiteral(string value, string options)
 | StringLiteral(string value)
 | SymbolLiteral(string value)

IntegerLiteral
 = BigIntLiteral() | Int16Literal() | Int31Literal() | Int32Literal()
 | Int64Literal() | Int8Literal() | UInt16Literal() | UInt31Literal()
 | UInt32Literal() | UInt64Literal() | UInt8Literal()

Type
 = BasicType()
 | WrapType(Type type)

WrapType
 = ArrayType() | ConstType() | GenericType() | PointerType()
 | ReferenceType() | StructType() | UnionType() | VolatileType()
```

図 26　ASDL を用いて記述した統合コードモデルの定義（続き 3）
Fig. 26　The unified code model in ASDL(Cont.3)