情報処理学会論文誌 Vol.54 No.2 945–960 (Feb. 2013)

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,a)] 大橋 昭[1,†1] 太田 大地[2] 鷲崎 弘宜[1] 深澤 良彰[1]

受付日 2012年5月15日, 採録日 2012年11月2日

概要：近年，プログラミング言語の多様化とソフトウェア開発を支援するソースコードの解析および変形ツールの開発が進んでいる．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，プログラミング言語とツール間に多対多の関係があり，すべての言語とツールの組合せに対して実装した場合，非常に莫大なコストが必要であるうえ，ツールごと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にくいという問題がある．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 を提案する．UNICOEN は言語共通な言語モデルを提供することで，ツールの開発コストおよび言語の追加コストを削減して，ツール間の差異を低減させる．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，その有用性を確認した．

キーワード：ソースコード解析，ソースコード変形，プログラミング言語，フレームワーク，ソフトウェアメトリクス

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

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

Received: May 15, 2012, Accepted: November 2, 2012

Abstract: Programming languages become more multifaceted and many analysis and transform tools for source code are being developed. However, high development costs are required to implement all combinations between programming languages and tools and there are differences of implementations and specifications in the tools because these tools support only one programming language. In this paper, we propose a framework for processing source code supporting multiple programming languages named UNICOEN. UNICOEN reduces development costs and prevents differences of implementations and specifications between tools. Conclusively, we evaluated UNICOEN developing 3 tools which supports 7 programming languages.

Keywords: source code analysis, source code transformation, programming language, framework, software metrics

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．たとえば，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 や Xtend 言語など新しい言語が次々と生まれている．

様々な言語が存在する中で，ソースコードを処理する

[1] 早稲田大学大学院基幹理工学研究科情報理工学専攻
Department of Computer Science and Engineering, Waseda University, Shinjuku, Tokyo 169–8555, Japan
[2] 株式会社 ACCESS
ACCESS CO., LTD., Chiyoda, Tokyo 101–0064, Japan
[†1] 現在，ソニー株式会社
Presently with Sony Corporation
[a)] kazuu@ruri.waseda.jp

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）処理系があげられる．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 [1]．

現在，多くのツールが言語ごとに開発されている．たとえば，静的解析ツールの FindBugs [2] は Java 言語のみに，JSLint [3] は JavaScript 言語のみに対応している．また，AOP 処理系の AspectJ [4] は Java 言語のみに，AOJS [5] 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．

ソースコードの解析や変形において対応言語間に共通処理が存在する．しかし，既存のフレームワークでは異なる言語間で処理を再利用する支援が不十分であるため，言語間の共通する処理が再利用されていない．したがって，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：複数の言語に対応するツールの開発に莫大な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．

本論文では，上述の問題を解決するため，UNICOEN（UNIfied source COde ENgineering framework）[*1] を提案する．UNICOEN は，解決 1：言語共通な統合コードモデルを提供して，解決 2：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．ソースコードの解析や変形を行うツール開発に UNICOEN を導入することで，ソースコード処理を統合コードモデル上のオブジェクト操作で実装でき，言語間で処理の再利用を行い，対応言語を追加するコストを低減させる．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を低減させて，問題 1 を緩和する．また，言語間で処理の再利用を行うことで，複数言語に対応したツール開発を支援する．このことは，対応言語それぞれに対して同等の機能を提供することを促し，対応言語間における仕様や機能の差異を抑えて，問題 2 を緩和する．

我々は，UNICOEN 上で 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．言語対応の追加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させるため，問題 1 と問題 2 を緩和できることを確認した．

本論文の主要な貢献は次の 3 点である．1) 7 種類の言語仕様から共通/相違点を整理した和集合を考えて統合コードモデルを設計した点，2) UNICOEN に言語対応を追加する利用者と UNICOEN 上でツールを開発する利用者向けに 2 種類の API を提供した点，3) 実際に UNICOEN に 7 種類の言語対応を追加して，3 種類のツールを実装して評価した点．

2. 既存ツールの問題点

2.1 問題 1：莫大な開発コスト

既存フレームワークでは異なる言語間でソースコード処理を再利用する支援が不十分であることから，ツールが言語ごとにばらばらに開発され，各言語で記載されたソースコードを処理する際に共通する処理が存在するにもかかわらず，再利用が行われていない．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るケースや，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，ツール開発者は別の言語にツールを対応させ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ったりするという問題が生じている．

たとえば，C 言語に対応した静的解析ツール Lint [6] 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 [7] 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．しかし，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．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．

2.2 問題 2：ツール間の差異

サーバクライアントモデルを採用するソフトウェアなど，複数の言語を利用するプロジェクトが増加している．Karus らの研究によると，オープンソースソフトウェアの開発に携わる開発者の多くは 4 種類以上の言語を利用している [8]．1 つのプロジェクトで複数の言語を利用している場合は，各言語に対して同じ解析をしたいにもかかわらず，言語ごとに作られたツールを組み合わせなければならないことがある．しかし，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．

さらに，言語の多様化によって採用可能な言語の選択肢

[*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．

が増えているため，開発者が携わるプロジェクトが変わると採用する言語も変わることが多い．プロジェクトの移動によって開発者が採用する言語が変化すると，以前のプロジェクトで利用していたツールを新しいプロジェクトで利用できない．そのため，採用言語に対応する新しいツールが必要となり，ツールの学習コストが必要になる．このようなコストはツールの利用を妨げる原因となる．

たとえば，テストカバレッジ測定ツール EMMA [9] は Java 言語に，Code coverage measurement for Python [10] 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．サーバクライアントモデルにより，サーバサイドを 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うえ，アスペクトの記述方法は AOP 処理系ごとに異なるため，ツール利用者は各記述方法を学ぶ必要がある．**図 1** の 1 から 7 行目と 9 から 17 行目で，すべ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示す．両者のアスペクトが同じ内容を示しているにもかかわらず，記述方法が大きく異なることが分かる．

```
1  public aspect Logger {
2   pointcut all() : execution(* *.*());
3   before() : all() {
4     System.out.println(thisJoinPoint.getSignature()
5       + " is executed.");
6   }
7  }
8
9  <?xml version="1.0" ?>
10 <aspectsetting>
11  <function functionname="/*" pointcut="execution">
12   <before>
13    <![CDATA[console
14      .log( __name__ + " is executed ."); ]]>
15   </before>
16  </function>
17 </aspectsetting>
```

図 1 メソッドの実行に対するロギングする AspectJ のアスペクトと AOJS のアスペクト

Fig. 1 A logging code for executing methods in AspectJ and AOJS.

3. UNICOEN の全体像

UNICOEN の全体像を**図 2** 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．UNICOEN はソースコードを構文解析して抽象構文木上で解析や変形をするツールの開発を支援する．また，UNICOEN はシンタックスに基づいてソースコードを構造化するため，特に，シンタックスに着目したり意味解析を完全に行わなかったりするようなツールの開発を助ける．

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 ライセンスを適用したオープンソースソフトウェアであり，Github からソースコードをダウンロードできる [11]．

UNICOEN は共通な抽象構文木（AST）の仕様として，解決 1：言語共通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共通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．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，同一言語内で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．

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）を言語ごと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，対応する言語の追加を支援する．UNICOEN は，解決 2：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，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追加コストの両方を低減させる．以上から，問題 1：複数の言語に対応するツールの開発に莫大なコストが必要である点を緩和する．

統合コードモデル上のオブジェクト操作でソースコード処理を実装することで，対応する言語間で共通する要素に

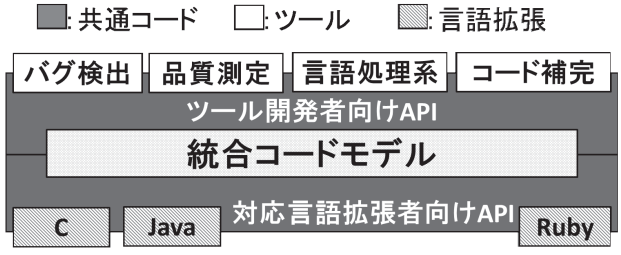

図 2 UNICOEN の全体像

Fig. 2 An overview of UNICOEN.

対する処理を共通化できる．このことは，複数の言語に対応したツールの開発を支援して，共通する要素に対する処理の内容を統一化することを促し，対応言語間における仕様や機能の差異を抑える．以上から，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緩和する．

UNICOENの利用者は，ツール開発者向けAPIを利用してツールを開発するツール開発者と，対応言語拡張者向けAPIを利用してUNICOENに対応言語を追加する対応言語拡張者の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，構文解析や一部の意味解析の処理の実装を省くことができる．さらに，言語共通な抽象構文木と共通のツール開発者向けAPIを利用して，UNICOENが対応するどんな言語に対しても，同じようなコードを記述してツールを開発できる．一方，対応言語拡張者は，OCマッパを開発することで，UNICOENが扱える言語の幅を広げられる．

4. UNICOENの詳細

本章では，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，様々な言語で記述されたソースコードを構造化する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．なお，各統合コードオブジェクトは親要素，子要素，ソースコード中の位置情報（行番号と列番号）を保有する．

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．たとえば，UNICOENは二項式の構文を認識するが，演算子の意味（たとえば，＋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7章で示す．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追加コストを低減させる．

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とることで統合コードモデルを設計した．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，それ以外を異なる要素として和集合をとった．

たとえば，while文の統合コードモデルを考える．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節を持ち，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持つ．したがって，和集合をとった結果，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．また，Javaのpackage宣言とC#のnamespace宣言は構文の見た目が異なるものの，構文が表す意味が等しいことに着目して，等しく名前空間を宣言する統合コードモデルとして扱っている．さらに，いくつかの言語では名前空間に直接メソッドやフィールドの定義を記述できるため，名前空間やクラス，インタフェースの宣言はすべて特殊なブロックの宣言として扱っている．

統合コードモデルを構築する際，共通要素を抽出するために，BNFにおける非終端記号の名前，構造，意味，記述可能な位置の類似性から共通かどうかを判断した．以下で，2種類の言語AとBについて，共通要素の候補を抽出する手順を説明する．

(1) 言語Aの最も抽象度の高い（プログラム全体に該当する）非終端記号を取り上げる．
(2) 取り上げた非終端記号と言語Bの候補になっていない非終端記号について，抽象度の高いものから，すなわち幅優先で比較して，以下のいずれかの条件を満たすものを探索する．ただし，取り上げた非終端記号の親に共通要素の候補が存在する場合は，言語B側の候補の非終端記号の子孫から先に幅優先で探索する．
   - 非終端記号の名前が類似している．
   - 非終端記号の子供の非終端記号どうしが類似していて，非終端記号の構造が類似している．
   - 非終端記号が表現する構文の意味や記述可能な位置が言語仕様書上で類似している．
(3) いずれかの条件を満たす要素が見つかった場合は共通要素の候補として記憶する．
(4) 取り上げた非終端記号が持つ子供の各非終端記号を幅優先で取り上げていき，手順(2)を繰り返す．取り上げられる非終端記号がなくなれば終了．

図3で非終端記号T1からT5を持つ言語AとT1'からT4'を持つ言語Bの共通要素の候補を抽出する様子を示す．図中の黒丸は非終端記号で白丸が終端記号で，T1とT1'，T2とT2'，T4とT4'，T5とT5'の組合せを共通要素の候補とする．まず，T1に対する共通要素の候補の探索を開始して，最初に比較したT1'が候補として見つかる．次に，T2に対する候補の探索を開始するが，T2の親T1はT1'と類似しているために，最初にT2'と比較を行

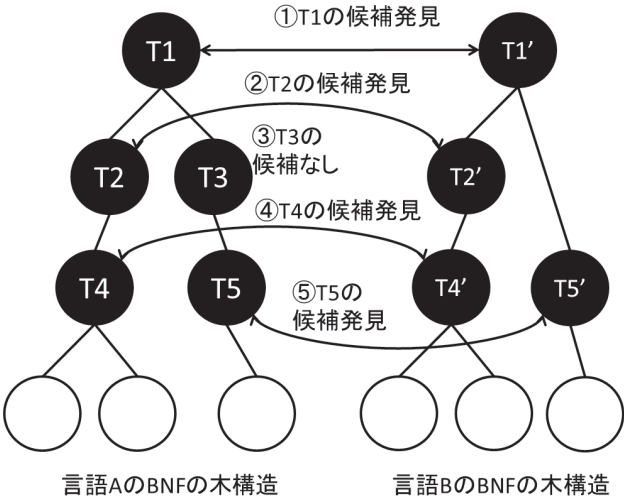

図 3 共通要素の候補を抽出する様子
Fig. 3 An image for selecting common elements.

```
Expression
  = If(Expression cond, Block body, Block else)
  | While(Expression cond, Block body, Block else)
  | DoWhile(Expression cond, Block body)
  | For(Expression init, Expression cond,
      Expression step, Block body, Block else)
  | FunctionDefinition(ModifierCollection mods,
      Type returnType, Identifier name,
      ParameterCollection parameters, Block body)
```

図 4 ASDL を用いて記述した統合コードモデルの定義の一部
Fig. 4 The part of unified code model in ASDL.

い候補として見つける．同様に T3 も T5’ と比較を行うが候補ではない．続けて T4’ と比較を行うがこれも候補でなく，候補を発見できない．その後，T4 に対する候補の探索を開始して，T4 の親 T2 は T2’ と類似しているために，最初に T4’ と比較を行い候補として見つける．最後に，T5 と T5’ が候補であることを発見して終了する．

以上の手順から共通要素の候補となっている非終端記号の組合せに対して，各言語の言語仕様書を読み直し，構文が持つ意味と記述可能な位置から，共通要素か否かを判断する．実際の UNICOEN の開発においては，早い段階から多くの言語の要素をモデル化するために，C，Java，C#，JavaScript の 4 種類の言語について上述の手順を拡張して同時に共通要素を抽出した．

Abstract Syntax Description Language（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 4** に示す[*2]．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．たとえば，Ruby 以外の 6 種類の言語で

[*2] 完全な定義は https://github.com/UnicoenProject/UNICOEN/blob/master/ASDL.txt で閲覧できる．

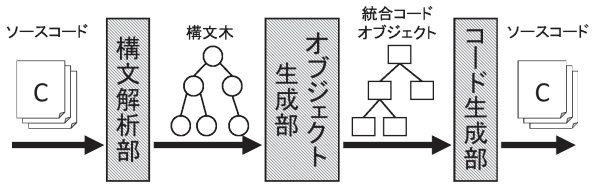

図 5 ソースコードと統合コードオブジェクトの相互変換
Fig. 5 A process of conversion and reverse conversion between source code and unified code objects.

```
var filePath = "code.java";
var ext = Path.GetExtension(filePath);
var progGen = UnifiedGenerators
  .GetProgramGeneratorByExtension(ext);
var uco = progGen.GenerateFromFile(filePath);
// 統合コードオブジェクトに対する処理を記述
var code = progGen.CodeGenerator.Generate(uco);
```

図 6 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. 6 A code to inter-convert between source code and unified code objects in C# with UNICOEN.

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と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．

4.2 対応言語拡張者向け API

UNICOEN は，対応言語の追加コストを低減させるために，対応言語拡張者向け API として，ANTLR [12] で構文解析した結果を.NET Framework 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．UNICOEN はツール開発者向け API となるインタフェースを提供するため，そのインタフェースに従って OC マッパを実装しなければならない．OC マッパの利用例として，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 5** と**図 6** で示す．

UNICOEN はソースコードから得られた統合コードオブジェクトについて，変換元と同じ言語のソースコードにセマンティクスを変えずに逆変換できることを保証する．対応言語拡張者はこの性質を満たすように OC マッパを開発しなければならないため，UNICOEN は上記の性質を保証していることを確認するためのテストコードを提供する．さらに，変換元の言語が持つ機能と構文の範囲で，統合コードオブジェクトを変更しても，コンパイル可能な状態

でその変更を反映したようなソースコードに逆変換できることを保証する．しかし，その範囲を越えて変更した場合や，機能や構文が異なる別の言語のソースコードへ変換する場合は，UNICOEN はその変換が正常に行えることを保証しない．なお，変換先の言語の OC マッパの開発者が個別に仕様を決めるため，たとえば，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た C 言語のソースコードに変換するような OC マッパを開発することができる．

OC マッパ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．我々は，ANTLR を用いて OC マッパ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 [13]．Code2Xml は ANTLR が生成したパーサを構文解析部として利用できるように自動修正する．さらに，オブジェクト生成部の実装を支援するために，XML の走査や式の解析処理などの共通処理を提供する．

以下に UNICOEN が対応する言語を追加する手順を述べる．1) 追加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．たとえば，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) すで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．たとえば，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 マッパを実装する．

なお，統合コードモデルを拡張しても，既存の OC マッパやツールは影響を受けない．統合コードモデルにクラスを追加した場合，そのクラスのインスタンスは既存の OC マッパからは生成されないが，ソースコード中に該当する構文が現れなかったのか，該当する構文がその言語に存在しないのか区別できない．同様に，プロパティを追加した場合，そのプロパティは既存の OC マッパでは null に初期化され，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，該当する要素がその言語に存在しないのか区別できない．

4.3 ツール開発者向け API

UNICOEN は，様々な言語向けのツールの開発コストを低減させるために，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，対応言語拡張者が言語ごとに実装した OC マッパによって実現される．**図 7** 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．`UnifiedElement` クラスは統合コードモデル上の要素を表すルートクラスである．また，`UnifiedProgramGenerator` と `UnifiedCodeGenerator` クラスは，ソースコードから統合コードオブジェクトに変換および逆変換する．

ツール開発者向け API は，`ModelSweeper` クラスの拡張メソッドとして実装することで，LINQ [14] と非常に高い親和性を持つ操作系として提供される．たとえば，**図 8** の 1 から 3 行目と 5 から 8 行目までのように，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，ツール開発者向け API を用いて if 文の数を表示するコードは非常に似ている．図 8 の変数 `ifElements` と `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，`IUnifiedElement` は統合コードオブジェクトの要素を表す．どちらも `IEnumerable` なので，LINQ が提供する拡張メソッド `Count()` を利用できる．

ツール開発者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を記述する．そのうえで，ソースコード解析ツールを作る場合は，統合コードオブジェクトに対して解析処理を記述する．一方，変形ツールを作る場合は，統合コードオブジェクトを変形してから，統合コードオブ

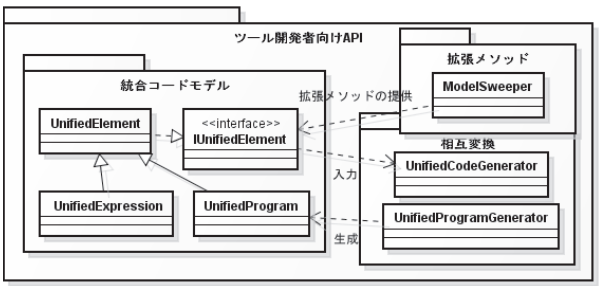

図 7 統合コードモデルとツール開発者向け API のクラス図
Fig. 7 A class diagram of the unified code model and the API for tool developers.

```
1 var xml = XDocument.Load("code.xml");
2 var es = xml.Descendants("if");
3 Console.WriteLine("#if elements: " + es.Count());
4
5 var uco = UnifiedGenerators
6   .GenerateProgramFromFile("code.java");
7 var ifs = uco.Descendants<UnifiedIf>();
8 Console.WriteLine("#ifs: " + ifs.Count());
```

図 8 if の要素ノード数と 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. 8 A code to enumerate 'if' elements and enumerate unified code objects of 'if' statement in C#.

```
var uco = UnifiedGenerators
  .GenerateProgramFromFile("code.java");
var count = uco.Descendants<UnifiedBlock>()
  .Sum(e => e.Count);
Console.WriteLine("#statements: " + count);
```

図 9 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. 9 A code to count statements in C# with UNICOEN.

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．たとえば，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すべて抽出して，その直下の子要素の数を数える．**図 9** でステートメント数を測定するコードを示す．

このように，ツール開発者向け API を利用して，統合コードオブジェクトを構成する要素を走査することで，任意の構文を容易に数え上げることができる．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したりするために，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 Remove メソッドを利用して子要素を追加，変更，削除したり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，様々な言語を対象として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．

4.4 UNICOEN を利用した開発に適したツール

統合コードモデルは 7 種類の言語それぞれについて構文要素の集合を作成し，それらの和集合をとったものから作成されている．そのため，統合コードモデルを構成する要素によっては，7 種類すべての言語が使用する要素もあれば，1 種類の言語のみが使用する要素もある．**表 1** に各要素を使用する言語の数ごとに集計した統合コードモデルの要素数を示す．さらに，付録にすべての要素と使用する言語の対応表を示す．なお，C#と Visual Basic 言語は同じ抽象構文木から統合コードモデルに変換しているため区別しない．

表 1 のように，複数の言語で使用される要素は 88 要素，単一の言語のみに使用される要素は 53 要素ある．たとえば，53 要素のうち C#言語の LINQ クエリ式に関する要素は 9 要素ある．LINQ クエリ式は.NET 言語特有の表現で他の言語と共有できないうえ，式の種類が多様で要素数も多い．そのほかに，C 言語の union 文，JavaScript 言語の with 文，Ruby 言語の redo 文，Python 言語のスライスがあげられる．これらの要素は統合コードモデルに含まれているが，他の言語から利用されない言語依存な要素である．なお，付録にすべての要素と各要素を使用する言語を示す．

表 1 使用する言語数ごとに集計した統合コードモデルの要素数

Table 1 The result of counting elements by the number of languages which use each element.

使用する言語数	6	5	4	3	2	1
要素数	31	16	4	14	23	53

UNICOEN は構文解析の結果に基づいて統合コードオブジェクトを生成するため，主に構文解析の情報から解析や変形処理を行うツールの開発に適している．さらに，ソースコード中に存在する要素のうち，ある特定の要素についてのみ操作するツールの開発に適している．特定要素のみを操作するツールは，それ以外の要素がどうであろうが影響を受けない．特に，言語共通な要素のみに着目する場合は，複数の言語に対応したツールを容易に開発できる．なぜなら，着目していない要素であれば，言語依存な要素であっても操作せずに無視できるためである．

5 章で述べる実際のツール開発の経験から，ソースコードの特定要素に着目して測定するメトリクスや，特定の要素の前後に新たな要素を挿入するアスペクト指向処理系の開発が容易であることが分かっている．たとえば，図 8 のように if 文の出現数を数える場合，if 文に該当する統合コードオブジェクトを数えればよいため，それ以外の要素に対する処理を記述する必要がなく，実装が容易である．また，ソースコードの特定部分のみを変形するテストカバレッジ測定やデバッガにも適している．さらに，構文解析結果から実装可能なバグパターン検出，コードクローン検出，コードフォーマッタなどのツール開発にも有効である．以上が UNICOEN による開発を想定するツールである．

一方，コンパイラやリファクタリングの処理系を実装する場合は，ソースコード中のすべての要素を解釈しなければならない．そのようなツールでは，言語非依存な要素と言語依存な要素の両方を解釈する必要があるので，言語依存な実装が必要である．さらに，これらのツールはソースコード全体について意味解析するため，言語ごとにその処理を実装する必要がある．UNICOEN は，ソースコード全体を解析して，かつ，意味論に踏み込むようなツール開発には不向きである．ただし，これらのツールを開発するにあたっても，構文解析の結果を共通した枠組みの中で処理できるという優位性がある．意味論に踏み込んで解析を行う既存のコンパイラフレームワークと比べれば劣るものの，既存のコンパイラコンパイラやパーサライブラリを利用して実装するよりは開発コストを抑えられる．

5. 評価

本章では UNICOEN の有用性を検証するために実際のツール開発を通した評価実験について述べる．UNICOEN

による対応言語の追加およびツール開発におけるコストの削減効果を示すために，我々は7種類の言語のOCマッパを実装して，その上で，メトリクス測定ツールを開発した．OCマッパおよびツールの実装に必要なステートメント数と，既存の言語処理系およびツールを構成するステートメント数を比較することで，UNICOENが開発コストを削減して，問題1を緩和することを確認する．さらに，ツール開発を繰り返すことで，UNICOENの実装に必要なステートメント数よりも，UNICOENによるステートメント数の削減の方が効果的であることを示す．

UNICOEN上で，複数の言語に対して同等の機能を提供する統一的なツールの開発が可能なことを示すために，メトリクス測定ツールとアスペクト指向プログラミング処理系を開発した．ツールが使用する統合コードモデルの要素について，言語間で処理を共通化することで，UNICOENが対応するすべての言語に対して各ツールが同様の機能を提供して，問題2を緩和することを確認する．

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マッパを実装した．実装したOCマッパ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．表中のOCマッパ項目は，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を表す．一方，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うえ，コンパイラ部分を分離できないため，処理系全体のステートメント数を提示する．また，Visual Basic処理系のソースコードがなかったため，表2には記載していない．

C，Java，JavaScript言語ではANTLR，C#とVisual Basic言語ではNRefactory [15]，Ruby言語ではruby_parser [16]，Python言語ではPython処理系の標準ライブラリのparserモジュールを利用している．ANTLRとNRefactoryはエラー回復機能を持っているため，構文エラーのあるソースコードからも統合コードオブジェクトを生成できる．たとえば，C言語においてステートメントの末尾にセミコロンをつけ忘れていても，その箇所をスキップしてパースを継続できる．一方，ruby_parserとparserモジュールはエラー回復機能を持っていないため，統合コードオブジェクトを生成できない．UNICOENはエラー回復機能の有効もしくは無効にするオプションを備えているが，実際にエラー回復機能を搭載するかどうかは，OCマッパの開発者が決定する．なお，C言語はプリプロセス後のプリプロセッサ命令を含んでいないソースコードを対象としている．そのため，6章でも示すように，プログラマが記述したソースコードを直接処理できない．

UNICOENは既存のソフトウェアを活用して，さらに，言語拡張者向け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を20分の1から50分の1に削減した．

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UNICOENを用いてMcCabeの複雑度や，CodeCity [17] が測定結果を街として可視化可能なメトリクスを測定するUniMetricsを開発した．

McCabeの複雑度を測定可能な既存ツールに，Java言語向けのSonar [18] とRuby言語向けのSaikuro [19] があげられる．しかし，これらの測定ツールの測定基準には拡張for文の扱いに相違点が存在するうえ，両者の測定結果を1つにまとめて表示できない．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．一方，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．また，CodeCityにはJava，C++，C#言語向けのメトリクス測定ツールがそれぞれ存在するが，複数の言語で開発

表 2 実装したOCマッパと既存の言語処理系のステートメント数の比較
（OCマッパの括弧内の数は既存の処理系に対するステートメント数の割合）
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 (The percents in OC mappers indicate the ratios of statements for existing processors).

言語	C	Java	C#	JavaScript	Python	Ruby
OCマッパ	727（4.86%）	1,003（7.85%）	399（1.08%）	626（1.64%）	636（4.13%）	501（3.49%）
既存の言語処理系	14,949	12,782	36,988	38,277	15,411	14,353

表 3 既存ツール Saikuro と 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 number of statements between Saikuro and UniMetrics.

	対応言語	ステートメント数
Saikuro	Ruby	321
UniMetrics	C，Java，C#，Visual Basic JavaScript，Python，Ruby	3

表 4 既存ツール PMCS と UniMetrics における対応言語と測定処理のステートメント数の比較

Table 4 A comparisons of the supported languages and the number of statements between PMCS and UniMetrics.

	対応言語	ステートメント数
PMCS	C#	1,478
UniMetrics	C，Java，C#，Visual Basic JavaScript，Python，Ruby	203

表 5 McCabe の複雑度の測定処理における統合コードモデルの要素と要素を共有している言語の一覧（L1: C，L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python）

Table 5 A table of elements in the unified code model for measuring McCabe complexity and languages (L1: C, L2: J ava, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
If	x	x	x	x	x	x
For	x	x	x	x		
Foreach		x	x	x	x	x
While	x	x	x	x	x	x
DoWhile	x	x	x	x	x	
Case	x	x	x	x	x	

されたプロジェクトから 1 つのメトリクス測定結果を生成して，可視化することができない．CodeCity はパッケージ，クラス，メソッドの構造ごとにコード行数，メソッド数，フィールド数の測定値を使用する．UniMetrics は，UNICOEN が対応する言語で記述されたソースコードを対象とでき，たとえば，Java と JavaScript 言語で開発された JsUnit のメトリクス測定結果を生成できる．

McCabe の複雑度を測定する処理のステートメント数について，Saikuro と UniMetrics の比較結果を**表 3** に示す．また，CodeCity 用の測定結果を生成する処理のステートメント数について，C#言語のみに対応する PMCS [20] と UniMetrics の比較結果を**表 4** に示す．また，**表 5** と**表 6** で，McCabe の複雑度の測定処理と，CodeCity 用の測定結果の生成処理における，統合コードモデルの要素とその要素を共通要素として使用している言語の一覧を示す．表

表 6 CodeCity 用の測定結果の生成処理における統合コードモデルの要素と要素を共有している言語の一覧（L1: C，L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python）

Table 6 A table of elements in the unified code model for generating the measurement result for CodeCity complexity and languages (L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
NamespaceDefinition		x	x			
ClassDefinition		x	x		x	x
FunctionDefinition	x	x	x	x	x	x
VariableIdentifier	x	x	x	x	x	x
Modifier	x	x	x	x	x	x

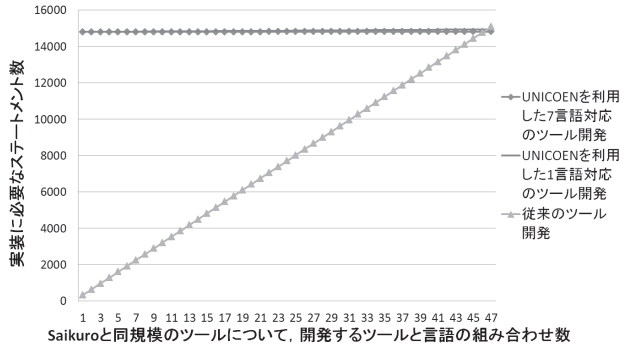

図 10 Saikuro と同規模のツールを開発する際に実装すべきステートメント数

Fig. 10 The number of statements for developing tools whose size is similar to the size of Saikuro.

中の記号 x は対応する列の言語が対応する行の要素を使用していることを示す．

Saikuro も PMCS も構文解析の処理を含んでおり，さらに，構文解析と同時に測定処理を行っている．一方，UNICOEN は OC マッパおよび統合コードモデルを提供することで，対応言語については統合コードオブジェクトの解析処理だけでツールを実装できる．表 5 と表 6 から，複数の言語に共通した統合コードモデルの要素を使用しており，対応言語間における共通要素に対する処理の再利用に成功している．その結果，表 3 と表 4 のように，UNICOEN がツール開発において実装すべきコード量を 7 分の 1 から 100 分の 1 程度削減した．

ライブラリや自動生成したソースコードを除き，OC マッパを含めて我々が実装した UNICOEN のステートメント数は 14796 である．**図 10** と**図 11** で，それぞれ Saikuro および PMCS と同規模で，同じように UNICOEN による支援が可能なツールを開発する場合に，実装に必要なステートメント数を比較したグラフを示す．横軸は言語とツールの組合せの数を示しており，言語追加をする場合でも同規模のツールを開発する場合でも 1 ずつ増加する．従来の

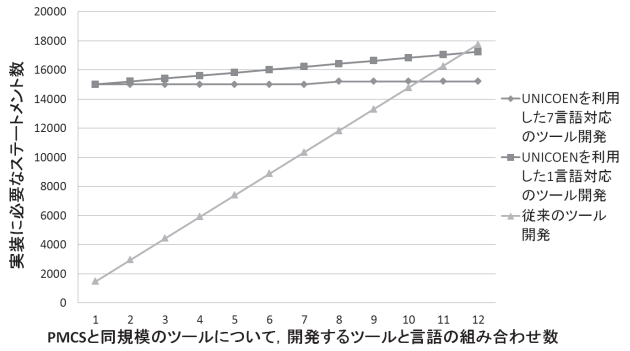

図 11 PMCS と同規模のツールを開発する際に実装すべきステートメント数

Fig. 11 The number of statements for developing tools whose size is similar to the size of PMCS.

ツール開発が UNICOEN を利用したツール開発を上回る組合せの数が現れるように横軸の値をとっている．ツールの実装に必要なステートメント数は，表 3 と表 4 の値を利用している．

従来のツール開発では，言語ごとに再利用せずにツールを開発するため，対応言語を追加する場合も同規模のツールを開発する場合でも，同じステートメント数が必要であると仮定する．一方，UNICOEN を利用した開発では，各ツールについて 7 言語すべてに対応する場合と，各ツールについて 1 言語のみに対応する場合の両方を示している．7 言語すべてに対応する場合は 7 種類の組合せを開発するたびに必要なステートメント数が増える．一方，1 言語のみに対応する場合は毎回必要なステートメント数が増える．

Saikuro のような小規模のツール開発において，UNICOEN の開発に必要なコストを回収するためには，47 種類の言語とツールの組合せについて実装を行う必要がある．一方，PMCS のような中規模のツール開発においては，7 言語すべてに対応する場合は 11 種類，1 言語のみに対応する場合は 12 種類の組合せについて開発を行う必要がある．したがって，開発するツールの規模によるものの，少なくとも 50 種類程度の組合せについて開発を行えば，UNICOEN のようなフレームワークを開発して再利用を促すことで，ツール開発に必要なコストを削減できることが分かる．

5.2.2 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．UniAspect では，ポイントカットを言語非依存に記述でき，アドバイスを織り込み先の言語ごとに記述する．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させて，より良いモジュール化を実現する．図 1 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 12** である．

UniAspect によるアスペクトが織り込まれる処理の流れ

```
1  aspect Logger {
2   pointcut allMethod() : execution(* *.*());
3
4   before : allMethod() {
5    @Java {
6      System.out.println(
7        JOINPOINT_NAME + " is executed.");
8    }end
9    @JavaScript {
10     console.log(JOINPOINT_NAME + " is executed.");
11   }end
12  }
13 }
```

図 12 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. 12 A logging code for executing methods in UniAspect.

表 7 UniAspect における統合コードモデルの要素と要素を共有している言語の一覧（L1: C，L2: Java，L3: C# and Visual Basic，L4: JavaScript，L5: Ruby，L6: Python）

Table 7 A table of elements in the unified code model for UniAspect complexity and languages (L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
Block	x	x	x	x	x	x
Call	x	x	x	x	x	x
Catch		x	x	x	x	x
FunctionDefinition	x	x	x	x	x	x
Return	x	x	x	x	x	x
VariableIdentifier	x	x	x	x	x	x
VariableDefinitionList	x	x		x		x
BinaryExpression	x	x	x	x	x	x
ClassDefinition		x	x		x	x
InterfaceDefinition		x	x			
EnumDefinition	x	x	x			

は次のとおりである．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あ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．上記のアスペクトの織り込み処理は，UNICOEN が提供する言語共通なツール開発者向け API を用いて実装できる．

表 7 に UniAspect が使用する統合コードモデルの要素と，その要素を共通要素として使用している言語の一覧を示す．また，表中の記号 x は対応する列の言語が対応する行の要素を使用していることを示す．UniAspect は AspectJ における call，execution，get，set，handler ポイントカットおよびインタータイプ宣言に対応している．表 7 のように各要素が複数の言語において有効であり，対応する言語間においてこれらの機能を再利用して実装することに成功

している．

5.1 節と 5.2.1 項から，以下の性質を備えるツールの開発については，問題 1 を緩和できることを確認した．

- 言語依存な要素の扱いが少ない．
- 意味解析を行わずに構文解析だけで実装できる．
- ソースコードに現れる特定の要素のみを操作する．
- 開発に必要なステートメント数が少なすぎない．

また，5.2.1 項と 5.2.2 項から，言語依存な要素が少なく，統合コードモデル上の一部の要素を処理するツールの開発については，UNICOEN が対応する言語に対して同等の機能を提供して，問題 2 を緩和できることを確認した．ただし，ソースコード全体にわたって意味解析を行うようなツールに関しては，さらなる支援が必要である．また，実際に複数の言語に対して同等の機能を提供するかどうかはツール開発者に依存するため，ツール開発者が統一的なツールを開発するように，さらなる支援が必要である．

6. 制限

UNICOEN には以下に示す制限がある．

- **静的解析に基づく統合コードオブジェクトの生成：** UNICOEN は静的な構文解析の結果に基づいて統合コードオブジェクトを生成している．そのため，得られた統合コードオブジェクトがどのような意味を持つか完全に識別しない．たとえば，コンパイラやリファクタリングのような，意味の解釈が必要なツールにおいては，統合コードモデル上で意味解析の処理を実装する必要がある．一般に，静的型付け言語においては，型システムの解釈を行うことで，静的にプログラムの意味を解釈することができる．一方，動的型付け言語，たとえば Ruby 言語においては，ある識別子が変数であるかメソッドであるかを静的に決定できず，プログラムを実行した際の文脈によって変数かメソッドか変化することもある．このような言語に対して，UNICOEN は既存のコンパイラコンパイラやパーサライブラリと同様に静的な構文解析処理の実装支援しか行わないため，利用者が動的解析を行うなどして，意味解析の処理を実装する必要がある．ただし，4.4 節および 5.2 節で述べたように，特定要素のみを操作するツールや，構文解析の結果に基づいて処理を行うツールについては，容易に実装可能である．
- **トークンの限定的な記憶：** UNICOEN はソースコード中のすべてのトークンを統合コードオブジェクトで記憶しない．たとえば，数式における括弧を抽象構文木で記憶しないように，統合コードモデルには括弧を記憶するための要素が存在しない．プログラマが記述したソースコード上のトークンを維持する形で，ソースコードを変形するような処理は記述できない．したがって，トークンを維持したリファクタリングやコードフォーマットを行うことは困難である．ただし，統合コードオブジェクトを介してソースコードの解析を行い，UNICOEN を利用せずに文字列処理で変形処理を実装することは可能である．
- **C 言語のプリプロセス：** C 言語においてはプリプロセス後のソースコードを対象としている．そのため，#define などのマクロが展開されたソースコードに対して処理する必要がある．プログラマが記述したソースコードを直接処理できないという制約はあるものの，マクロの利用が限定的であれば他の言語と同じように処理を行えるうえ，マクロによって隠蔽されている潜在的なソースコードの性質を扱うことが可能である．
- **統合コードモデルの構築における共通要素の定性的な判断：** 非終端要素の名前，構造，意味，記述可能な位置の 4 つの観点から，要素の組合せが共通要素であるかどうか判断する．しかし，これらの判断は定性的に行われており，機械的に共通要素を抽出できない．ただし，著者らが構築した統合コードモデル上でツールの開発に成功しているため，たとえば，名前であれば文字列の類似度などを算出することで，現状の判断結果から定量的に判断する基準を新たに作成できる．

7. 関連研究

表 8 で UNICOEN と本章で述べる既存ツールやフレームワークの比較結果を示す．言語追加は対応言語の追加支援の有無，共通モデルは共通の言語モデルの有無，解析は解析処理の実装支援の有無，変形は変形処理の実装支援の有無を示す．表中の記号 x が記載されている項目は支援があることを示す．

Lattner ら [21] はコンパイラフレームワーク LLVM を提案した．類似したソフトウェアとして，Java VM や.NET Framework など中間言語の実行環境と GCC があげられる．これらのソフトウェアは，完全に意味解析をして中間言語に変換するため，対応言語を追加するコストが非常に大きい．さらに，中間言語はマシン語に近い非常に抽象度の低い仕様であり，多くの場合，シンタックス情報を得られ

表 8 UNICOEN と既存ツールやフレームワークの比較表
Table 8 A table for comparing UNICOEN with existing tools and frameworks.

フレームワーク	言語追加	共通モデル	解析	変形
UNICOEN	x	x	x	x
LLVM	x	x	x	x
MASU		x	x	
Sapid			x	x
srcML		x	x	x
DMS	x		x	x
TXL	x			x
Stratego	x			x

ない．そのため，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．また，評価の項目で実装したようなソースコードの処理ツールは完全な意味解析を必要としない．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．UNICOEN は統合コードモデルと開発者向け API を提供することで，どの言語に対しても同じようにツールを開発できる．また，UNICOEN では，必要に応じて統合コードモデル上で意味解析を行うことで，ツールに必要な処理を実装できる．したがって，意味解析を完全に必要とはしないツール開発を対象としたとき，UNICOEN は既存ソフトウェアより優位性がある．

Higo ら [22] 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，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 が対応する言語すべてに対応しているうえ，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．また，MASU は言語非依存な抽象構文木の提供にとどまっているが，UNICOEN は対応言語拡張者向け API を提供することで対応言語の追加を支援する．そのうえ，ソースコードの解析処理だけでなく，変形処理も言語非依存に記述可能である．

Fukuyasu ら [23] はソースコードを解析した結果を実体および関連としてモデル化したデータベースを提供する CASE ツール・プラットフォーム Spaid を提案した．Sapid は C，Java，JavaScript，JSP，HTML 言語に対応している．対応言語によって解析処理の内容が異なり，Java に対応する Japid では，意味解析の情報を利用してセマンティクスを維持したプログラムの変形が可能である．一方，Sapid は新しい言語に対応するための仕組みを備えておらず，UNICOEN は対応言語の拡張性において優位性がある．

Collard ら [24] は XML フォーマットで抽象構文木のノード情報をソースコードに埋め込むツール srcML を提案した．C，C++，Java 言語に対応しており，言語間で共通した抽象構文木のスキーマーを利用する．また，XML 文書からソースコードに逆変換するツールも備えており，変形処理も実装できる．解析処理は DOM や SAX など，変形処理は XSLT など既存の XML 技術を適用できる．一方，srcML は新しい言語に対応するための仕組みを備えておらず，UNICOEN は対応言語の拡張性において優位性がある．

Baxter ら [25] が提案する DMS は，ソースコードの解析や変形処理を実装するためのツールキットである．DMS は構文解析器を備えており，文法ファイルを与えることで対応言語を追加できる．また，ソースコード書き換えルールを記述するための言語および言語処理系を提供しており，ソースコードの解析と変形の両方の処理を実装できる．さらに，汎用的な意味解析処理を提供しており，コントロールフローやデータフローの生成が可能である．DMS は UNICOEN が対応する言語のうち Ruby を除いてすべてに対応しているうえ，COBOL や FORTRAN など様々な言語の文法ファイルが用意されている．一方，UNICOEN のような共通の言語モデルが用意されていないため，UNICOEN は言語間での処理の再利用性において優位性がある．

Cordy [26] が提案する TXL は，ソースコードの変形処理を記述するためのプログラミング言語および言語処理系を提供する．TXL を利用することでソースコードから任意の文章へのマッピング処理を記述できる．TXL は構文解析器を備えており，文法ファイルを与えることで対応言語を追加できる．多くの文法ファイルが公開されており，UNICOEN が対応する言語のうち Ruby を除いてすべてに対応しているうえ，Ada や Delphi など様々な言語の文法ファイルが用意されている．一方，TXL は変形処理に特化しているうえ，UNICOEN のような共通の言語モデルが用意されていないため，UNICOEN は解析処理の実装支援および言語間での処理の再利用性において優位性がある．

Bravenboer ら [27] が提案する Stratego は，ソースコードの変形処理を記述するためのプログラミング言語および言語処理系を提供する．Stratego は TXL と異なりソースコードからソースコードへの変換処理を対象としており，メタプログラミングやプログラムの最適化などで利用可能である．Stratego は AspectJ，Java，Jimple，PHP4，PHP5 言語に対応している．Stategо は構文解析器を備えており，SDF 文法ファイルを与えることで対応言語を追加できる．一方，Stategо は変形処理に特化しているうえ，UNICOEN のような共通の言語モデルが用意されていないため，UNICOEN は解析処理の実装支援および言語間での処理の再利用性において優位性がある．

8. 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，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，構文解析を行い抽象構文木上でソースコードの解析や変形をするツールの開発を支援する．我々は，UNICOEN に C，Java，C#，Visual Basic，JavaScript，Python，Ruby 言語の対応を追加して，そのうえで，これらの言語に対応したメトリクス測定ツール，CodeCity の言語対応の追加，AOP 処理系を開発した．そして，対応言語の追加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，意味解析を完全に行わないようなツールの開発において，統合コードモデルとツール開発者向けAPIを利用することで実装に必要な記述量を大幅に低減させることできること，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させることを確認した．以上から，UNICOENが問題1および問題2を緩和できることを確認した．

今後の展望として，次のような点があげられる．

- 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へ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だけでなく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．
- 現状では，UNICOENの対応言語を追加する際，人手でOCマッパを開発する必要がある．さらに，新たな言語を追加する際に，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追加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．
- UNICOENは構文解析の結果に基づいて統合コードオブジェクトを生成することにとどめているが，変数のスコープの解決や関数の呼び出し先と呼び出し元の関係の解決など，対応する言語間において共通する意味解析の処理が存在する．したがって，共通の意味解析処理を実装することで，適用可能なツールの幅を広げることが考えられる．
- UNICOENを利用して複数の言語に対して同様の機能を提供するツールを開発するかどうかは開発者次第である．ドキュメントや統合開発環境で表示されるソースコード中のコメントなどの内容を充実させることで，統合コードモデル上の要素がどの言語について共通して利用可能かどうか示すことで，複数の言語に対応した統一的なツールの開発を促すことが考えられる．

謝辞 UNICOENの開発にあたり岩澤宏希氏から多大なる協力を受けた．心より感謝を申し上げる．本研究の一部は，独立行政法人情報処理推進機構の2010年度未踏IT人材発掘・育成事業「複数言語対応のソースコード処理ツールのフレームワークと利用例」による助成を受けた．本研究の一部は，一般財団法人安藤研究所の安藤博記念学術奨励賞による助成を受けた．また，本研究の一部は，ローム株式会社による研究助成を受けた．

参考文献

[1] 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 (2006).

[2] Hovemeyer, D. and Pugh, W.: Finding bugs is easy, *SIGPLAN Not.*, Vol.39, pp.92–106 (2004).

[3] Crockford, D.: JSLint, The JavaScript Code Quality Tool, available from ⟨http://www.jslint.com/⟩ (accessed 2012-02-04).

[4] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proc. 15th European Conference on Object-Oriented Programming, ECOOP '01*, London, UK, UK, pp.327–353, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=646158.680006⟩ (2001).

[5] 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, *Proc. 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS '09*, New York, NY, USA, ACM, pp.31–36 (2009).

[6] Johnson, S.C.: Lint, a C Program Checker, *COMP. SCI. TECH. REP.*, pp.78–88 (1978).

[7] Logilab: pylint 0.25.1: Python Package Index, available from ⟨http://pypi.python.org/pypi/pylint⟩ (accessed 2012-02-04).

[8] Karus, S. and Gall, H.: A study of language usage evolution in open source software, *Proc. 8th Working Conference on Mining Software Repositories, MSR '11*, New York, NY, USA, ACM, pp.13–22 (online), DOI: 10.1145/1985441.1985447 (2011).

[9] Roubtsov, V.: EMMA: A free Java code coverage tool, available from ⟨http://emma.sourceforge.net/⟩ (accessed 2012-11-27).

[10] Batchelder, N.: coverage 3.5.3: Python Package Index, available from ⟨http://pypi.python.org/pypi/coverage⟩ (accessed 2012-11-27).

[11] 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).

[12] Parr, T.J. and Quong, R.W.: ANTLR: A predicated-LL (k) parser generator, *Softw. Pract. Exper.*, Vol.25, pp.789–810 (1995).

[13] Sakamoto, K.: Code2Xml, available from ⟨https://github.com/exKAZUu/Code2Xml/⟩ (accessed 2012-12-21).

[14] Microsoft: LINQ (Language-Integrated Query), available from ⟨http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx⟩ (accessed 2012-11-15).

[15] Krüger, M.: icsharpcode/NRefactory - GitHub, available from ⟨https://github.com/icsharpcode/NRefactory⟩ (accessed 2012-08-24).

[16] Ryan Davis, E.H.: RubyForge: ParseTree – ruby parse tree tools: Project Info, available from ⟨http://rubyforge.org/projects/parsetree/⟩ (accessed 2012-08-24).

[17] Wettel, R., Lanza, M. and Robbes, R.: Software systems as cities: A controlled experiment, *Proc. 33rd International Conference on Software Engineering, ICSE '11*,

New York, NY, USA, ACM, pp.551–560 (2011).
[18] SonarSource: Sonar, available from ⟨http://www.sonarsource.org/⟩ (accessed 2012-02-04).
[19] Blut, Z.: Saikuro: A Cyclomatic Complexity Analyzer, available from ⟨http://saikuro.rubyforge.org/⟩ (accessed 2012-02-04).
[20] Doernenburg, E.: erikdoe / PMCS / overview - Bitbucket, available from ⟨https://bitbucket.org/erikdoe/pmcs/⟩ (accessed 2012-02-04).
[21] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. International Symposium on Code Generation and optimization: Feedback-Directed and Runtime Optimization, CGO '04*, Washington, DC, USA, IEEE Computer Society, pp.75–86 (online), available from ⟨http://dl.acm.org/citation.cfm?id=977395.977673⟩ (2004).
[22] Higo, Y., Saitoh, A., Yamada, G., Miyake, T., Kusumoto, S. and Inoue, K.: A Pluggable Tool for Measuring Software Metrics from Source Code, *Proc. 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 (2011).
[23] Fukuyasu, N., Yamamoto, S. and Agusa, K.: CASE Tool Platform Sapid Based on a Fine Grained Repository (Special Issue on Parallel Processing), *IPSJ Journal*, Vol.39, No.6, pp.1990–1998 (1998).
[24] Collard, M.L., Decker, M.J. and Maletic, J.I.: Lightweight Transformation and Fact Extraction with the srcML Toolkit, *Proc. 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, SCAM '11*, Washington, DC, USA, IEEE Computer Society, pp.173–184 (online), DOI: 10.1109/SCAM.2011.19 (2011).
[25] Baxter, I.D., Pidgeon, C. and Mehlich, M.: DMS (R): Program Transformations for Practical Scalable Software Evolution, *Proc. 26th International Conference on Software Engineering, ICSE '04*, Washington, DC, USA, IEEE Computer Society, pp.625–634 (2004).
[26] Cordy, J.R.: The TXL source transformation language, *Sci. Comput. Program.*, Vol.61, No.3, pp.190–210 (online), DOI: 10.1016/j.scico.2006.04.002 (2006).
[27] Bravenboer, M., Kalleberg, K.T., Vermaas, R. and Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation, *Sci. Comput. Program.*, Vol.72, No.1-2, pp.52–70 (online), DOI: 10.1016/j.scico.2007.11.003 (2008).

付　　録

A.1 統合コードモデルの要素と使用する言語

表 A·1，**A·2**，**A·3** に統合コードモデルの要素と使用する言語の一覧を示す．表中の記号 x は対応する列の言語が対応する行の要素を使用していることを示す．なお，表では具象クラスとして実装されている要素のみを掲載している．

表 A·1 統合コードモデルの要素と使用する言語の一覧（その 1）（L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby，L6: Python）
Table A·1 A table of elements in the unified code model and languages (L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
Program	x	x	x	x	x	x
Block	x	x	x	x	x	x
Comment	x	x	x	x	x	x
VariableDefinition	x	x	x	x		x
VariableDefinitionList	x	x		x		x
ClassDefinition		x	x		x	x
AnnotationDefinition		x				
EigenClassDefinition					x	
EnumDefinition	x	x	x			
InterfaceDefinition		x	x			
ModuleDefinition					x	
NamespaceDefinition		x	x			
StructDefinition	x		x			
UnionDefinition	x					
EventDefinition			x			
FunctionDefinition	x	x	x	x	x	x
PropertyDefinition			x			
PropertyDefinitionPart			x			
Constructor		x	x	x		
InstanceInitializer				x		
StaticInitializer		x	x			
TypeConstrain		x	x			
SuperConstrain		x	x			
ReferenceConstrain			x			
ImplementsConstrain		x	x			
ExtendConstrain		x	x			
EigenConstrain					x	
ConstructorConstrain			x			
AnnotationCollection		x				
ArgumentCollection	x	x	x	x	x	x
CaseCollection	x	x	x		x	
CatchCollection		x	x	x	x	x
ExpressionCollection	x	x	x	x	x	x
GenericArgumentCollection		x	x			
GenericParameterCollection		x	x			
IdentifierCollection	x	x	x	x	x	x
ModifierCollection	x	x	x	x	x	x
OrderByKeyCollection			x			
ParameterCollection	x	x	x	x	x	x
TypeCollection	x	x	x		x	x
TypeConstrainCollection	x	x	x		x	x
Modifier	x	x	x	x	x	x
Annotation		x	x			
Parameter	x	x	x	x	x	x
Argument	x	x	x	x	x	x
GenericParameter		x	x			
GenericArgument		x	x			

表 A·2 統合コードモデルの要素と使用する言語の一覧（その 2）（L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby，L6: Python）

Table A·2 A table of elements in the unified code model and languages (L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
Call	x	x	x	x	x	x
New	x	x	x	x		
Property	x	x	x	x	x	x
Indexer	x	x	x	x	x	x
Slice						x
Identifier	x	x	x	x	x	x
VariableIdentifier	x	x	x	x	x	x
LabelIdentifier	x		x			
SuperIdentifier		x	x			x
ThisIdentifier		x	x			x
TypeIdentifier			x			
VaueIdentifier			x			
Cast	x	x	x			
Type	x	x	x	x	x	x
ArrayType	x	x	x			
ConstType	x					
GenericType		x	x			
PointerType	x					
StructType	x		x			
UnionType	x					
VolatileType	x					
NullLiteral	x	x	x	x	x	x
BooleanLiteral		x	x	x	x	x
CharLiteral	x	x	x	x		x
FractionLiteral	x	x	x		x	x
RegularExpressionLiteral				x	x	
StringLiteral	x	x	x	x	x	x
SymbolLiteral					x	
IntegerLiteral	x	x	x	x	x	x
ArrayLiteral		x	x	x	x	x
IterableLiteral						x
ListLiteral				x		x
MapLiteral				x	x	x
SetLiteral						x
TupleLiteral						x
KeyValue				x	x	x
Range					x	
BinaryExpression	x	x	x	x	x	x
TernaryExpression	x	x	x	x	x	x
UnaryExpression	x	x	x	x	x	x
BinaryOperator	x	x	x	x	x	x
UnaryOperator	x	x	x	x	x	x
Sizeof	x		x			
Typeof		x	x			
If	x	x	x	x	x	x
For	x	x	x	x		x
Foreach		x	x	x	x	x

表 A·3 統合コードモデルの要素と使用する言語の一覧（その 3）（L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby，L6: Python）

Table A·3 A table of elements in the unified code model and languages (L1: C, L2: Java, L3: C# and Visual Basic, L4: JavaScript, L5: Ruby, L6: Python).

	言語					
要素	L1	L2	L3	L4	L5	L6
While	x	x	x	x	x	x
DoWhile	x	x	x	x	x	
Switch	x	x	x	x	x	
Case	x	x	x	x	x	
Label	x	x		x		
Lambda			x	x		x
Proc					x	
Try		x	x	x	x	x
Catch		x	x	x	x	x
Fix			x			
Synchronized		x	x			
Using			x			
With				x		
Break	x	x	x	x	x	x
Continue	x	x	x	x	x	x
Return	x	x	x	x	x	x
Throw		x	x	x	x	x
Goto	x		x			
Redo					x	
Retry					x	
YieldBreak			x			
YieldReturn			x		x	x
Alias					x	
Assert		x		x		x
Default			x			
Defined					x	
Delete				x	x	x
Exec						x
Import		x	x			x
Pass				x		x
Print						x
PrintChevron						x
StringConversion						x
MapComprehension						x
IterableComprehension						x
ListComprehension						x
SetComprehension						x
LinqExpression			x			
LinqQuery			x			
FromQuery			x			
GroupByQuery			x			
JoinQuery			x			
LetQuery			x			
OrderByQuery			x			
SelectQuery			x			
WhereQuery			x			
OrderByKey			x			

坂本 一憲 （学生会員）

2009年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．2010年同大学大学院基幹理工学研究科修士課程修了．2010年より同大学院基幹理工学研究科博士課程に所属．2011年より同大学助手．現在に至る．日本ソフトウェア科学会，ACM各学生会員．

大橋 昭

2010年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．2010年より同大学大学院基幹理工学研究科修士課程に所属．2012年よりソニー株式会社に所属．現在に至る．

太田 大地

2009年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．2011年同大学大学院基幹理工学研究科修士課程修了．2011年より株式会社ACCESSに所属．現在に至る．

鷲崎 弘宜 （正会員）

2003年早稲田大学大学院博士後期課程修了，博士（情報科学）．同大学助手，国立情報学研究所助手を経て，2008年より同大学理工学術院准教授，同研究所客員准教授．再利用と品質保証を中心にソフトウェア工学の研究や教育に従事．情報処理学会代表会員，情報規格調査会SC7/WG20小委員会主査，日科技連SQiP研究会運営委員長．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM各会員．

深澤 良彰 （正会員）

1976年早稲田大学理工学部電気工学科卒業．1983年同大学大学院博士課程修了．同年相模工業大学工学部情報工学科専任講師．1987年早稲田大学理工学部助教授．1992年同教授．工学博士．主として，ソフトウェア再利用技術を中心としたソフトウェア工学の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM各会員．