この資料のPDF版を下記のURLに置いています
http://goo.gl/h8noJP

コードクローン研究の これまでとこれから

大阪大学　大学院情報科学研究科
higo@ist.osaka-u.ac.jp
肥後 芳樹

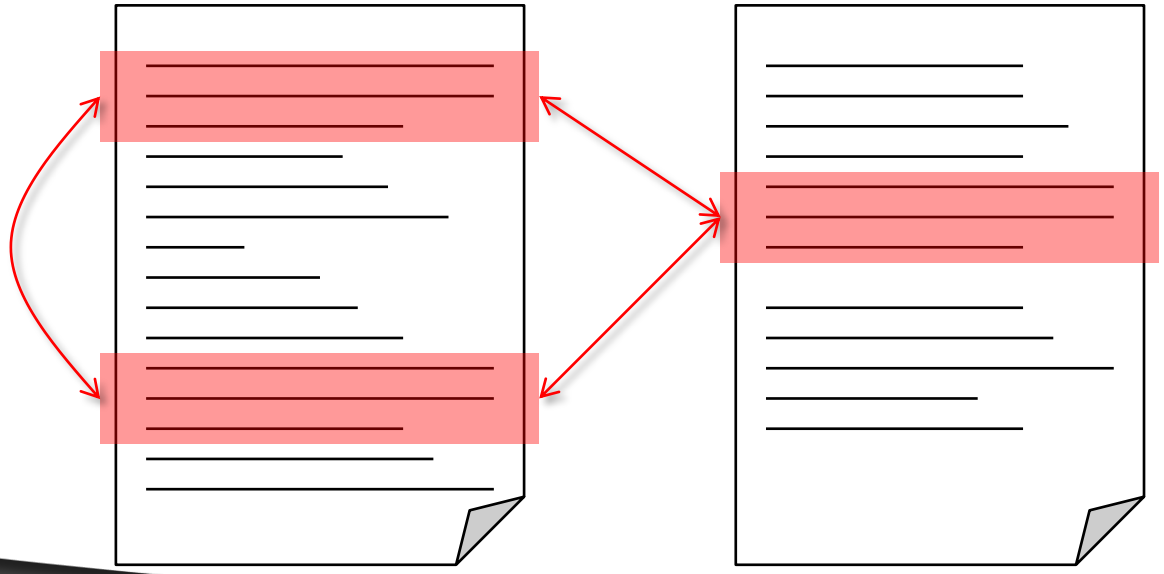
本チュートリアルの構成

- 第一部：コードクローンとは？
- 第二部：これまでの研究
- 第三部：CCFinderといくつかの適用事例
- 第四部：コードクローン検出手法の例
- 第五部：同じ機能を持つコードを見つけるための調査
- 第六部：これからに向けて

第一部
コードクローンとは？

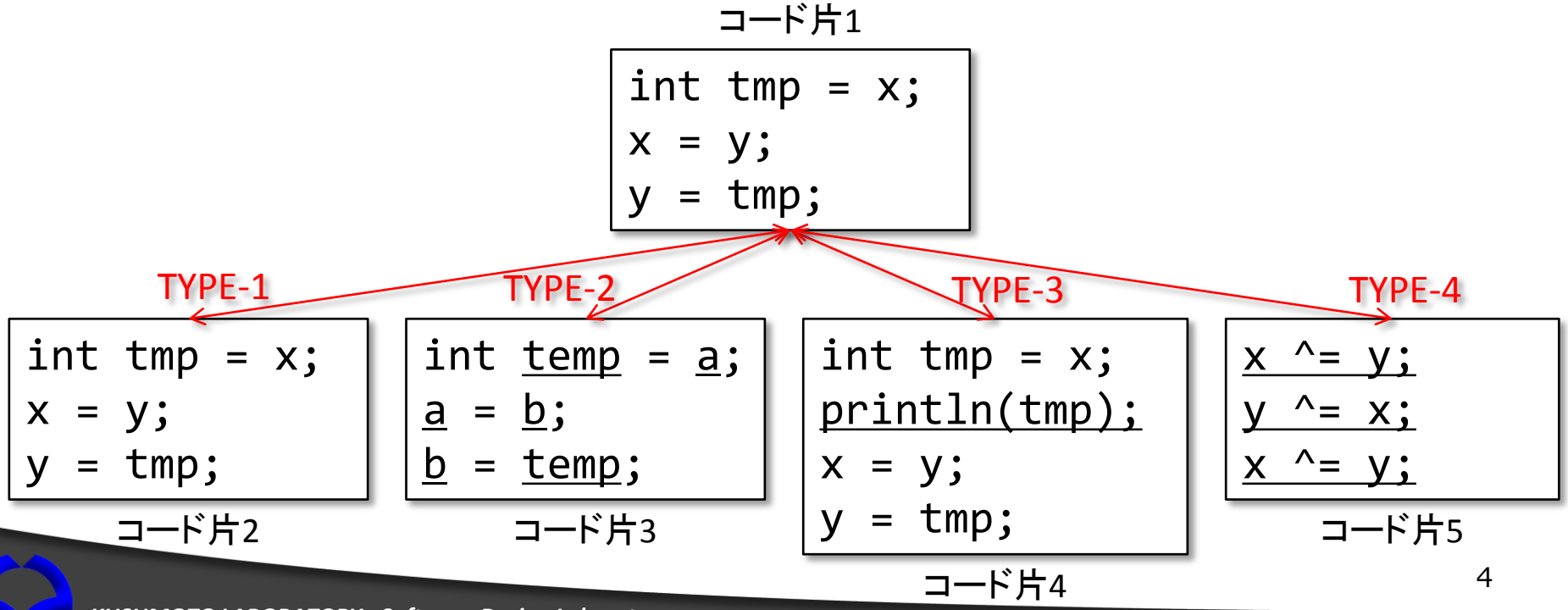
コードクローンとは？

- ソースコード中の類似したコード片
  - 「類似」の厳密かつ一般的な定義は存在しない
  - 各検出手法では，「類似」を定義し，その定義に基づいてコードクローンの検出を行っている
  - 短く「クローン」ということが多い



コードクローンの種類

- TYPE-1：空白やタブ等のコーディングスタイルを除くと完全一致
- TYPE-2：変数名やリテラル等の字句単位での違いを含む
- TYPE-3：字句単位よりも大きな違いを含む
- TYPE-4：同じ入力に対しては同じ出力を返す
  - 私はTYPE-4をコードのクローンとは言いたくない



コードクローンの発生理由

- **開発技術を原因とするもの**
  - コード生成ツール
  - プログラミング言語の機能不足
  - 定形処理・横断的関心事
- **開発者の行為を原因とするもの**
  - コピーアンドペーストによる再利用
  - リファクタリングの欠如
  - 設計段階での共通機能の括り出し失敗
  - コード剽窃
- **開発組織の方針を原因とするもの**
  - 「動いているコードに触るな！」
  - 管理の欠如（短すぎる納期等）
  - 行数による評価

コードクローンは悪もの？

- （20年前）全てのクローンは悪，できるだけ取り除くべき！！

- （現在）開発・保守を阻害するのは一部のクローンのみ
  - 多くの研究により，同時修正を必要とするクローンの割合は少ないことがわかっている

    ほとんどがオープンソースに対する実験
    商用ソフトウェアについては不明

ソフトウェア開発の失敗例(特許庁)

特許庁の情報システム開発の失敗により約55億円の損失が発生2012年1月

特許庁情報システムに関する技術検証委員会
第1回議事要旨(一部抜粋)

＜　　からのヒアリング＞

○ 残件の発生原因として、共通したものはあるか。

○ 通常、プログラミングでは「クローンの弊害」とかよく言われていて、クローンは作るべきではないのだが、このプロジェクトでは、設計段階でクローンを非常にたくさん作ってしまい、明らかにクローンの弊害が出ている。つまり、既に検証された部分をコピーペーストしてクローンを作るのではなくて、品質が検証される前の段階でクローンをつくってしまっているので、元の設計箇所に問題があると当然そのクローンの分だけ修正対象箇所が増えていく。

○ そもそもユーザアプリケーションを相互に疎な関係として作りたいという考え方があり、これは特許庁の要望であった。そういう関係を実現しようとすると、同じようなソフトウェアが作られることになるがそういうことかと確認したところ、ＴＳＯＬの回答は、コピーペーストするという話であった。そこで、修正が必要になったらどうするか聞いたら、それは大もとを直して、同じことを他の部分でもやれば修正できる、という回答をもらった記憶がある。普通に考

逆説的に考えると…

- コピペ後に同時修正を必要としないコードクローンは，開発や保守に悪影響はない
- コピペは必要な機能を瞬時に実装できる素晴らしい（？）再利用技術
  - コピペ皆無の開発はありえない

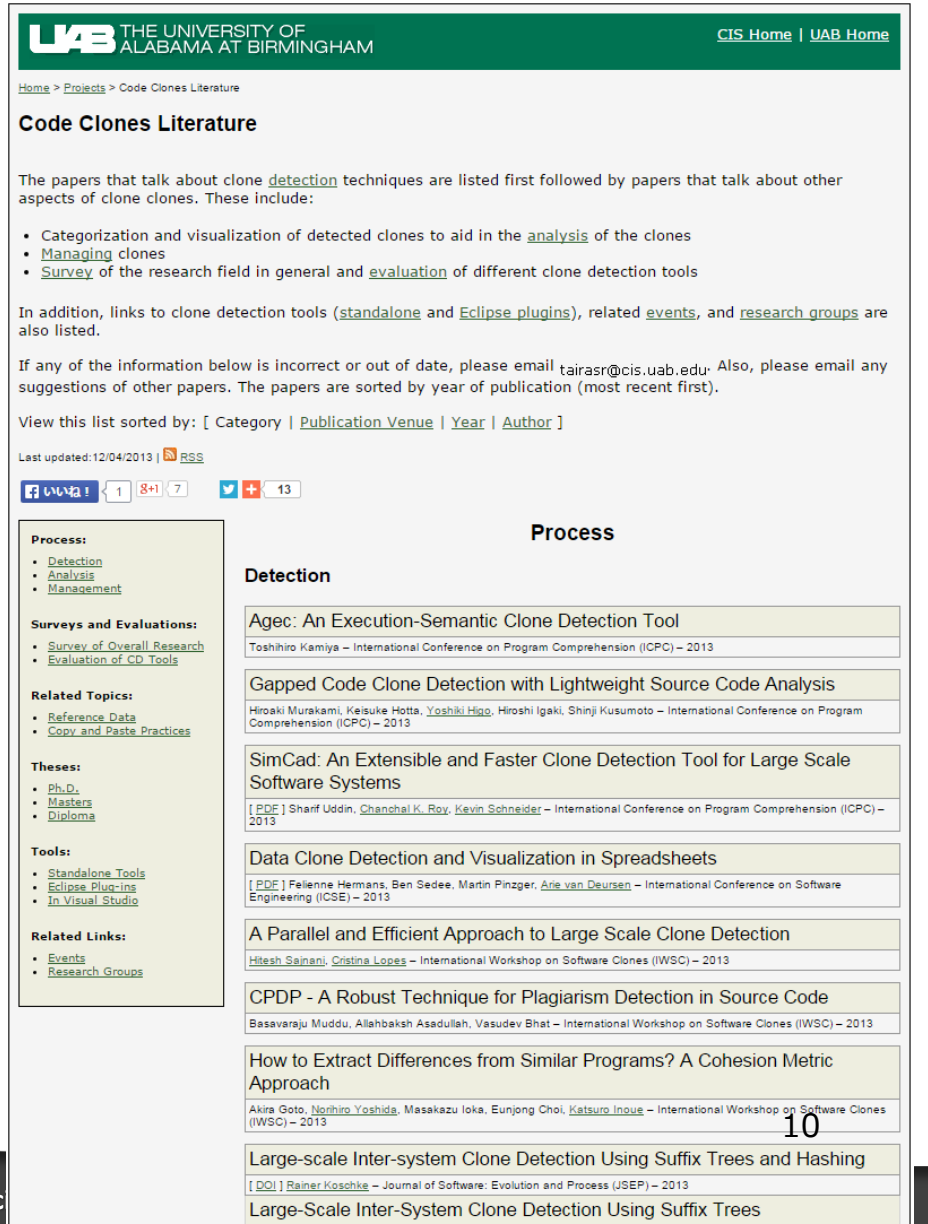
コピペをするときには，コピペ元コードの信頼性が担保されていることを確認すべき

第二部
これまでの研究

Code Clones Literature

http://students.cis.uab.edu/tairasr/clones/literature/

- コードクローン関連の論文をまとめているサイト
- 1994から2013年（途中）までの文献をリスト化
- 文献はいくつかのカテゴリに分けられて整理
  - 検出
  - 分析
  - 管理
  - サーベイ＆ツール評価



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

CIS Home | UAB Home

Home > Projects > Code Clones Literature

Code Clones Literature

The papers that talk about clone detection techniques are listed first followed by papers that talk about other aspects of clone clones. These include:

- Categorization and visualization of detected clones to aid in the analysis of the clones
- Managing clones
- Survey of the research field in general and evaluation of different clone detection tools

In addition, links to clone detection tools (standalone and Eclipse plugins), related events, and research groups are also listed.

If any of the information below is incorrect or out of date, please email tairasr@cis.uab.edu. Also, please email any suggestions of other papers. The papers are sorted by year of publication (most recent first).

View this list sorted by: [Category | Publication Venue | Year | Author]

Last updated:12/04/2013 | RSS

いいね！ 1 | 8+1 7 | 13

Process:
- Detection
- Analysis
- Management

Surveys and Evaluations:
- Survey of Overall Research
- Evaluation of CD Tools

Related Topics:
- Reference Data
- Copy and Paste Practices

Theses:
- Ph.D.
- Masters
- Diploma

Tools:
- Standalone Tools
- Eclipse Plug-ins
- In Visual Studio

Related Links:
- Events
- Research Groups

Process

Detection

Agec: An Execution-Semantic Clone Detection Tool
Toshihiro Kamiya – International Conference on Program Comprehension (ICPC) – 2013

Gapped Code Clone Detection with Lightweight Source Code Analysis
Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, Shinji Kusumoto – International Conference on Program Comprehension (ICPC) – 2013

SimCad: An Extensible and Faster Clone Detection Tool for Large Scale Software Systems
[PDF] Sharif Uddin, Chanchal K. Roy, Kevin Schneider – International Conference on Program Comprehension (ICPC) – 2013

Data Clone Detection and Visualization in Spreadsheets
[PDF] Felienne Hermans, Ben Sedee, Martin Pinzger, Arie van Deursen – International Conference on Software Engineering (ICSE) – 2013

A Parallel and Efficient Approach to Large Scale Clone Detection
Hitesh Sajnani, Cristina Lopes – International Workshop on Software Clones (IWSC) – 2013

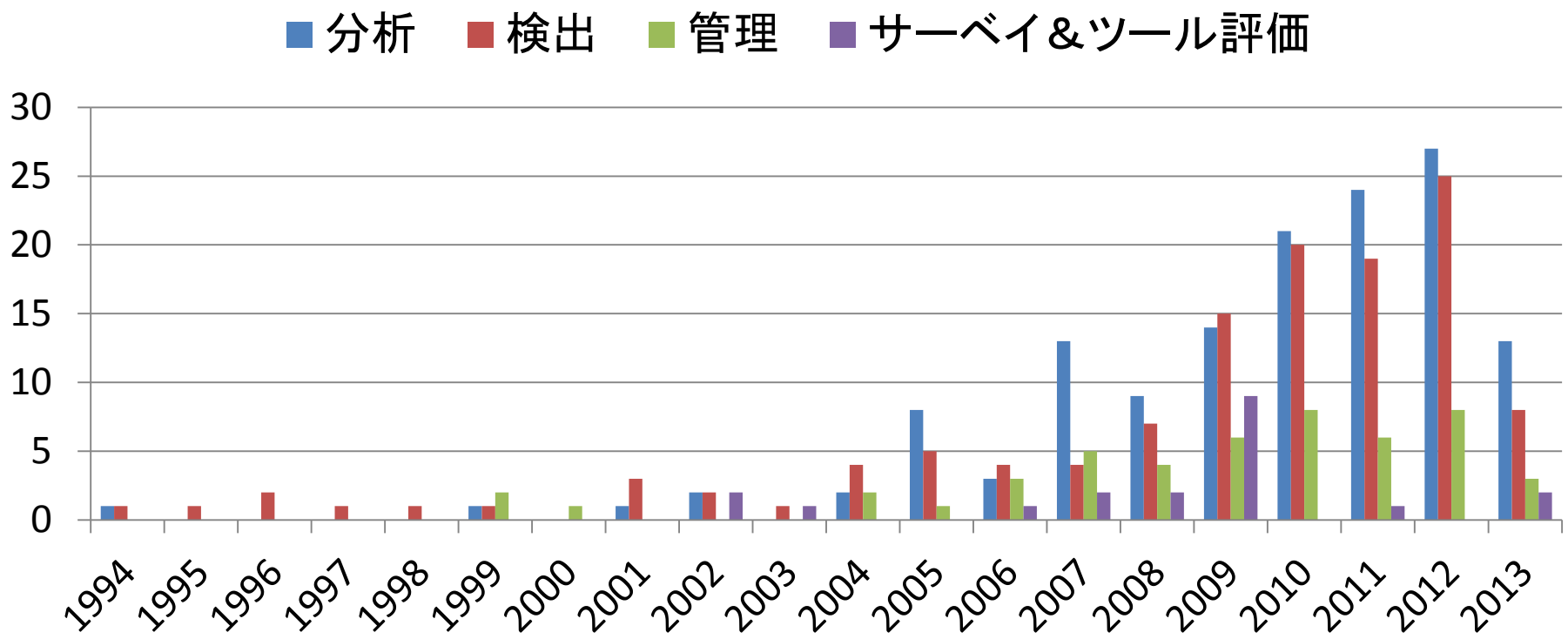
CPDP - A Robust Technique for Plagiarism Detection in Source Code
Basavaraju Muddu, Allahbaksh Asadullah, Vasudev Bhat – International Workshop on Software Clones (IWSC) – 2013

How to Extract Differences from Similar Programs? A Cohesion Metric Approach
Akira Goto, Norihiro Yoshida, Masakazu Ioka, Eunjong Choi, Katsuro Inoue – International Workshop on Software Clones (IWSC) – 2013

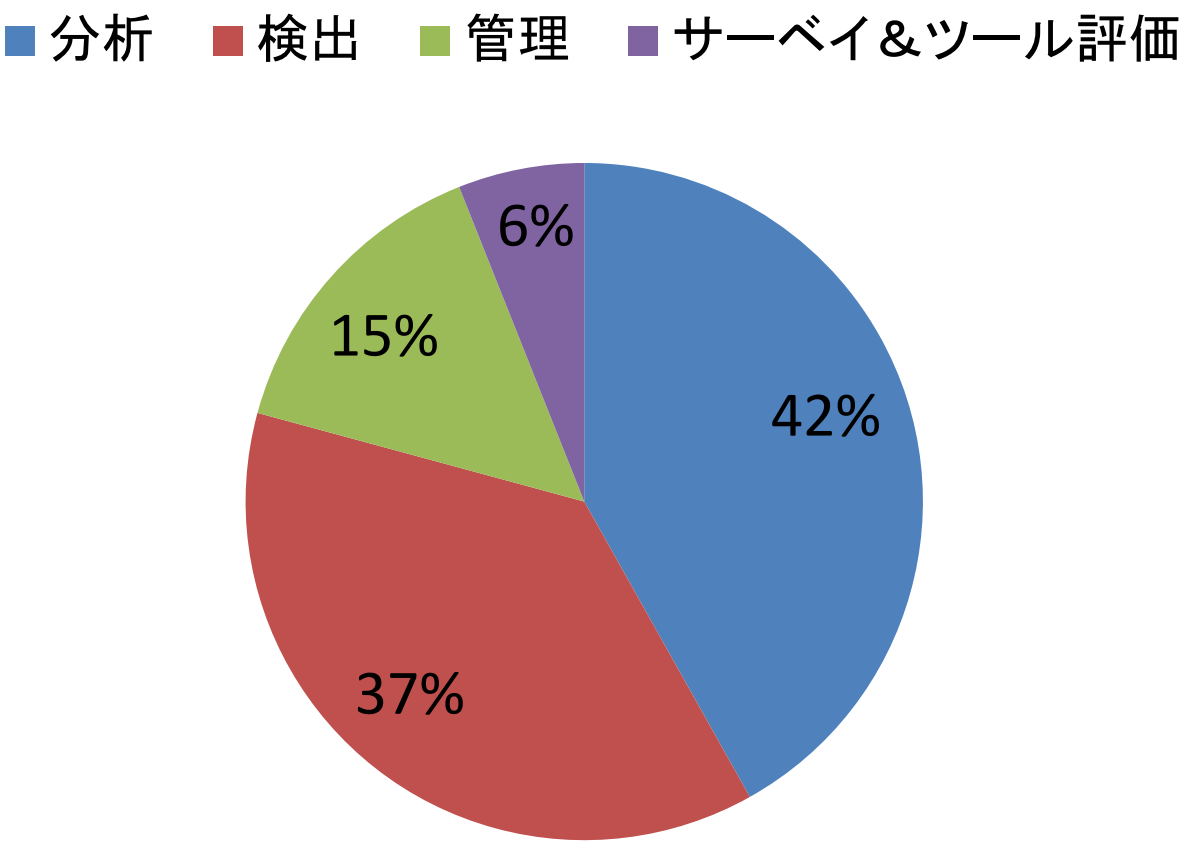
Large-scale Inter-system Clone Detection Using Suffix Trees and Hashing
[DOI] Rainer Koschke – Journal of Software: Evolution and Process (JSEP) – 2013

Large-Scale Inter-System Clone Detection Using Suffix Trees

文献数の推移


分析
検出
管理
サーベイ＆ツール評価
30
25
20
15
10
5
0
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013

各カテゴリの割合



International Workshop on Software Clones

- コードクローンに関する研究の国際ワークショップ
  - これまでに9回開催
  - 10回目はSANER2016のワークショップとして開催？
- コードクローンの研究者がほぼ集まる
- 採択されるのはそんなに難しくない
  - 採択率は50%程度

9th INTERNATIONAL WORKSHOP ON SOFTWARE CLONES

IWSC 2015

Friday March 6, 2015, Montreal, Canada

In conjunction with SANER'15

Welcome
Objectives
Committee
Program Committee
Call for Papers
Important Dates
Schedule
Keynote

WE ARE PROUDLY ANNOUNCING THAT IWSC 2015 IS CO-LOCATED WITH SANER!

PROGRAM SCHEDULE IS NOW AVAILABLE. CHECK THE SCHEDULE LINK ON THIS PAGE

いいね！ 9人がいいね！しています。

Tweets Follow

IWSC 2015 @IWSC2015 20 Jan
The workshop is going to be held on 06 March, 2015 with #SANER15.

IWSC 2015 @IWSC2015 20 Jan
The notifications have just been sent out. IWSC'15 co-located with #SANER15 is going to be an exciting workshop with lots of new ideas.
Expand

Welcome to IWSC'15

Software clones are often a result of copying and pasting as an act of ad-hoc reuse by programmers, and can occur at many levels, from simple statement sequences to blocks, methods, classes, source files, subsystems, models, architectures and entire designs, and in all software artifacts (code, models, requirements or architecture documentation, etc.). Software clone research is of high relevance for software engineering research and practice today.

Topics of interest include, but are not limited to:

- Use cases of clone management in the software lifecycle
- Experiences with clone management in practice
- Types, distribution, and nature of clones in software systems
- Causes and effects of clones
- Techniques and algorithms for clone detection, analysis, and management
- Clone and clone patterns visualization
- Tools and systems for detecting software clones
- Applications of clone detection and analysis
- System architecture and clones
- Effect of clones to system complexity and quality
- Clone analysis in families of similar systems
- Measures of code similarity
- Economic and trade-off models for clone removal
- Evaluation and benchmarking of clone detection methods

大阪大学(井上研・楠本研)

- 多数の論文誌・国際会議
  - T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28, 7 (July 2002), 654-670.
    - 2015/8/31現在の被引用数:1087(google scholar)
- 受賞
  - 平成27年度科学技術分野の文部科学大臣表彰　科学技術賞　研究分野
  - 第35回市村学術賞貢献賞
  - 情報処理学会論文賞(2007年, 2014年)
  - その他多数
- 博士号取得への貢献
  - がっつりクローンネタ:7人
  - クローンを要素技術として利用:4人

コードクローンについて勉強するために
—日本語のサーベイ論文・解説論文—

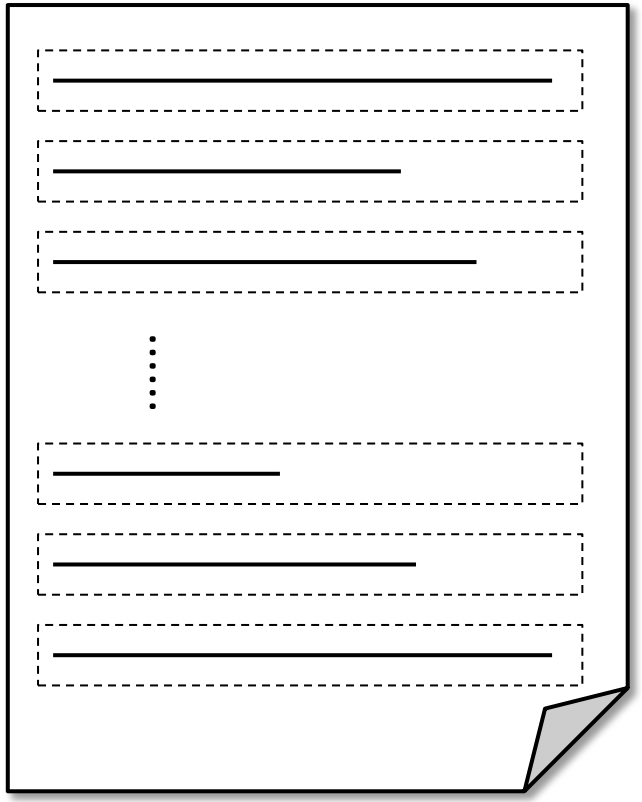
- 堀田圭佑, 肥後芳樹, 楠本真二, “生成抑止, 分析効率化, 不具合検出を中心としたコードクローン管理支援技術に関する研究動向”, コンピュータソフトウェア, Vol.31, No.1, pp.14–29, 2014年2月.
- 肥後芳樹, 吉田則裕, “コードクローンを対象としたリファクタリング”, コンピュータソフトウェア, Vol.28, No.4, pp.43–56, 2011年11月.
- 神谷年洋, 肥後芳樹, 吉田則裕, “コードクローン検出技術の展開”, コンピュータソフトウェア, Vol.28, No.3, pp.29–42, 2011年8月.
- 肥後芳樹, 楠本真二, 井上克郎, “コードクローン検出とその関連技術”, 電子情報通信学会論文誌D, Vol.91–D, No.6, pp.1465–1481, 2008年6月.

基本となる検出技術

- 行単位での検出
- 字句単位での検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやフィンガープリントなど，そのほかの技術を用いた検出

行単位での検出

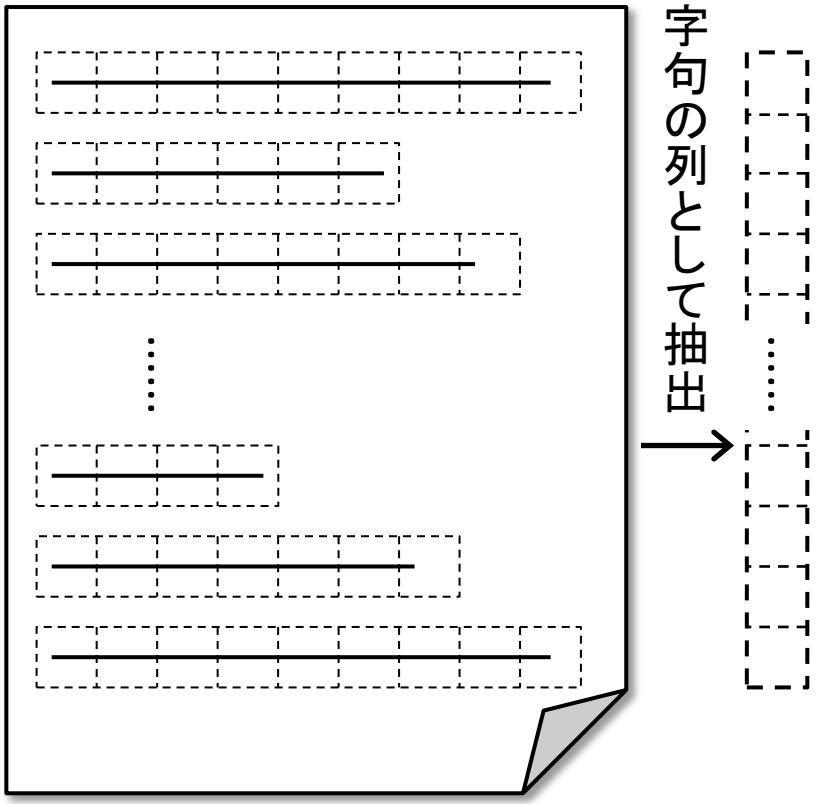
- ソースコードを行単位で比較することにより，クローンを検出
  - しきい値以上連続して重複している行がクローンとして検出される
- 他の検出技術に比べ検出速度が高速
- 同じ処理を行っているコードであってもコーディングスタイルの違う場合はクローンとして検出できない



ソースコードの行そのものがクローン検出の単位

字句単位での検出

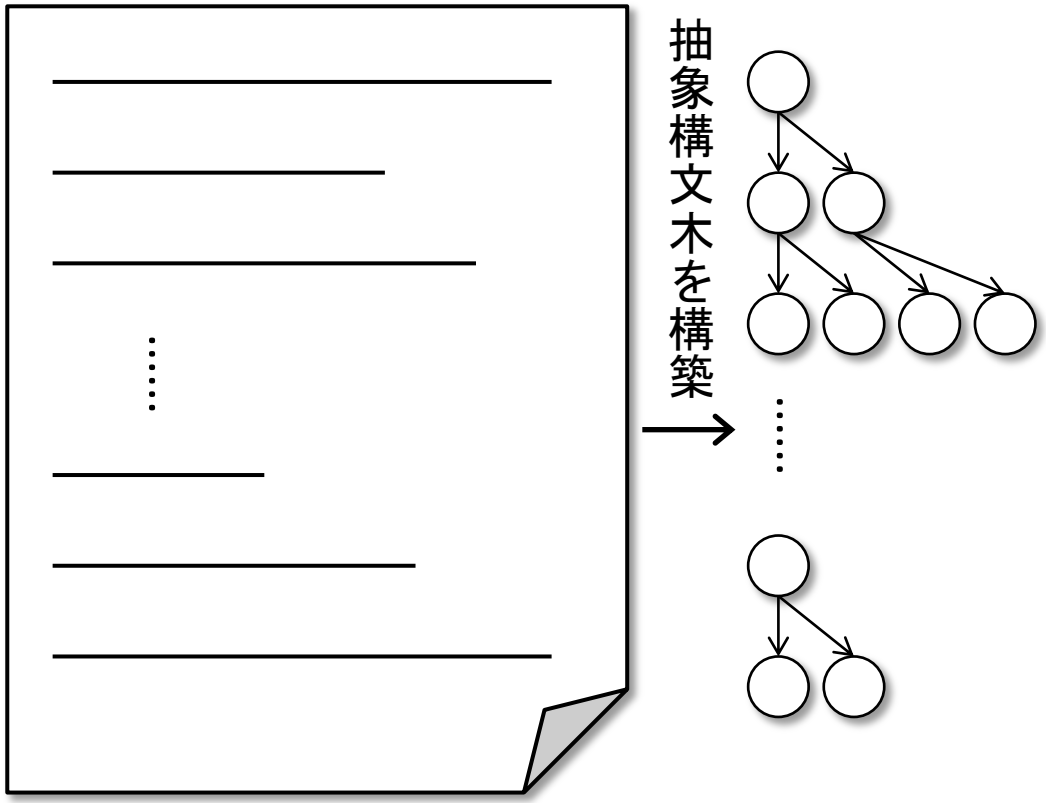
- 前処理として，ソースコードは字句の列に変換される
  - しきい値以上連続して一致する部分列がクローンとして検出される
- 抽象構文木やプログラム依存グラフを用いた検出に比べると高速
- 検出結果がコーディングスタイルに依存することはない



連続して一致する部分列がクローンとして検出される

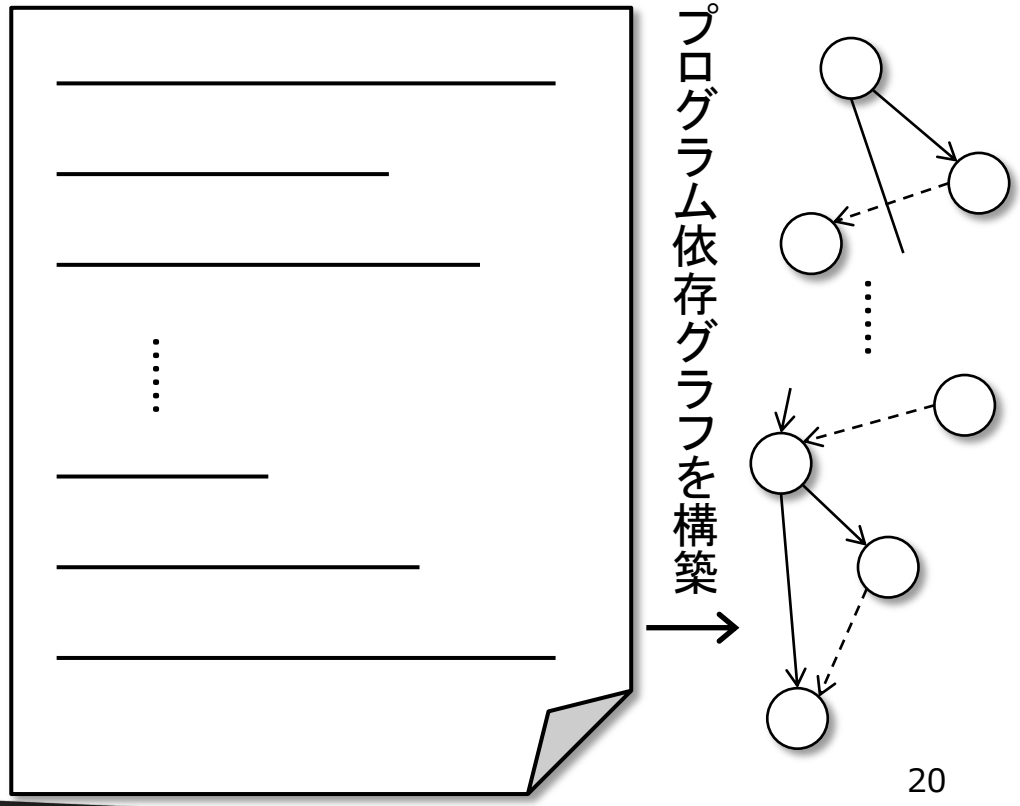
抽象構文木を用いた抽出

- ソースコードに対して構文解析を行い，抽象構文木を構築
  - 同形の部分木がクローンとして検出される
  - 検出されるクローンは構造的なまとまりを持つ
- 検出結果がコーディングスタイルに依存することはない
- 検出に必要な時間的・空間的なコストは高い



プログラム依存グラフを用いた検出

- ソースコードに対して意味解析を行い，要素（文や式）間の依存関係を抽出
  - 要素を頂点，依存関係を有向辺とするグラフ
  - 同形部分グラフがクローンとして検出される
- 特殊なクローンの検出が可能
  - 巻き付きクローン
  - 順序入れ替わりクローン
- 検出コストが非常に高い



巻き付きクローンと順序入れ替わりクローン

巻き付き(intertwined)クローン

```
    ...
++  tmpa = UCHAR(*a),
--  tmpb = UCHAR(*b);
++  while (blanks[tmpa])
++      tmpa = UCHAR(*++a);
--  while (blanks[tmpb])
--      tmpb = UCHAR(*++b);
++  if (tmpa == '-') {
        tmpa = UCHAR(*++a);
        ...
    }
--  else if (tmpb == '-') {
        if(...UCHAR(*++b)...) ...
```

順序入れ替わり(reordered)クローン

```
    fp3 = lookaheadset + tokensetsize;
    for (i = lookaheads(state);
         i < k; i++) {
++      fp1 = LA +
++              i * tokensetsize;
++      fp2 = lookaheadset;
++      while (fp2 < fp3)
++          *fp2++ |= *fp1++;
    }
```

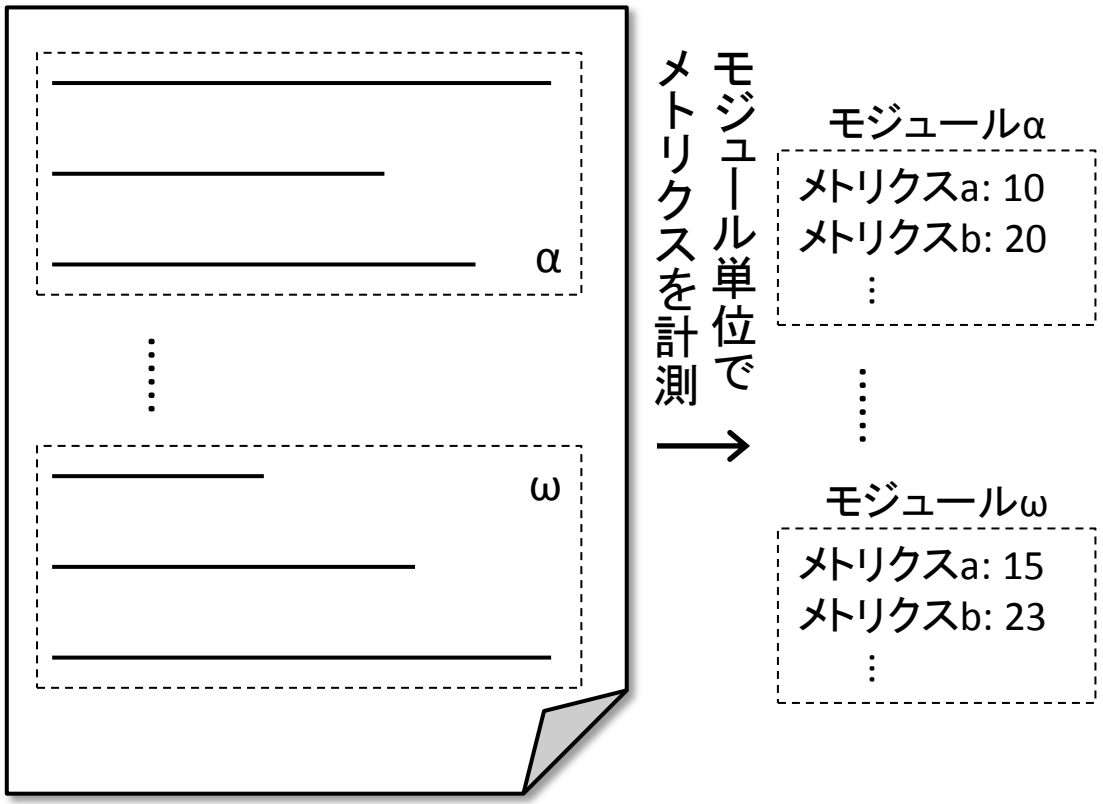
```
    fp3 = base + tokensetsize;
    ...
    if (rp) {
        while ((j = *rp++) >= 0) {
            ...
++          fp1 = lookaheadset;
++          fp2 = LA +
++                  j * tokensetsize;
++          while (fp1 < fp3)
++              *fp1++ |= *fp2++;
        }
    }
```

その他の技術を用いた検出

- プログラムのモジュール（ファイル，クラス，メソッド等）に対してメトリクスを計測する
  - メトリクス値が近いモジュールがクローンとして検出される

- サイズの小さいモジュールはメトリクス値に差がでないため誤検出の可能性が高くなる

- サイズの大きなモジュールはその一部が類似していてもクローンとして判定されない



基本技術の特徴と発展方法

	TYPE-1	TYPE-2	TYPE-3	検出速度
行単位	○	×	×	激速
字句単位	○	○	×	速い
抽象構文木	○	○	×	遅い
プログラム依存グラフ	○	○	○	激遅
メトリクス	○	○	○	速い

- 種々の検出技術は基本技術を発展／組み合わせたもの
  - 検出能力の向上
  - 検出速度の向上
  - 人間が必要としないクローンの検出防止

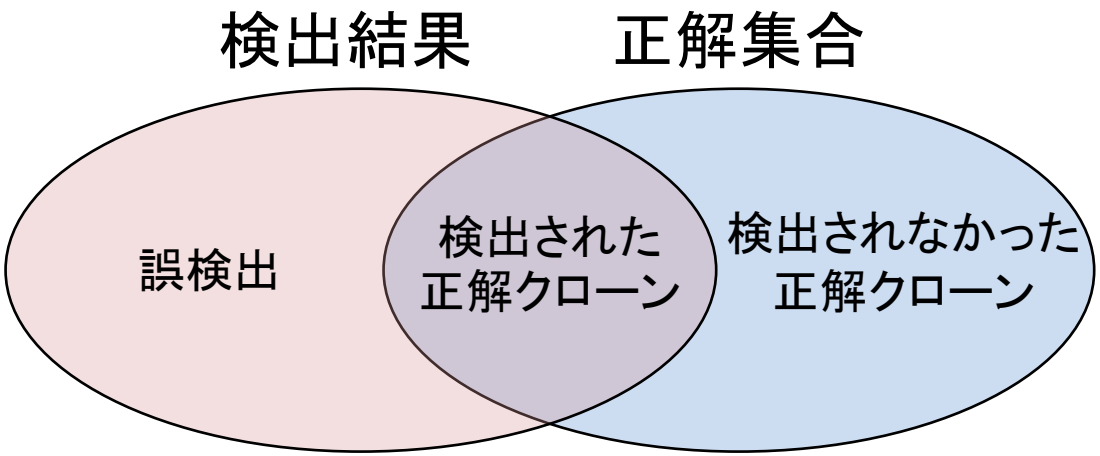
コードクローン検出技術の比較・評価

- さまざまな検出手法が提案されたために，それらを比較・評価をする手法が必要
  - Bellonら[1]：定量的な比較・評価
  - Royら[2]：定性的な比較・評価

[1] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.* 33, 9 (September 2007)
[2] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* 74, 7 (May 2009)

Bellonらの比較・評価：適合率と再現率

- クローンの正解集合を作成することにより，各検出ツールの適合率・再現率を算出
  - 適合率：検出結果のどれだけが正解か
  - 再現率：正解クローンをどの程度検出できているか
    - OK値とGOOD値→参考資料



$$\text{適合率} = \frac{\text{検出された正解クローン}}{\text{検出結果}}$$

$$\text{再現率} = \frac{\text{検出された正解クローン}}{\text{正解集合}}$$

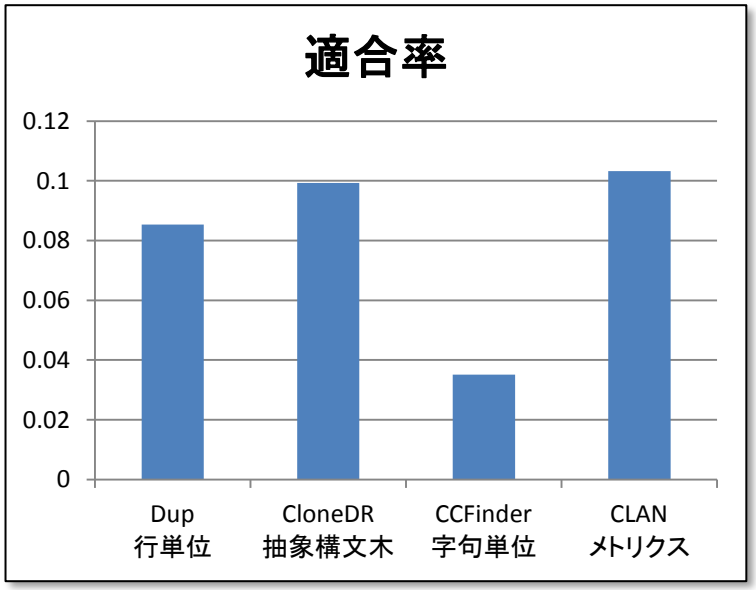
Bellonらの比較・評価：正解集合

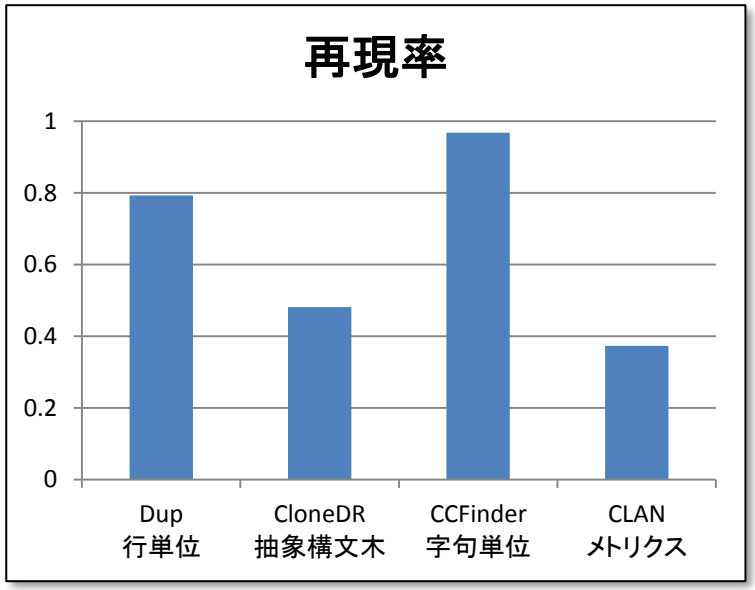
- ソフトウェアに含まれる全ての正解クローンを知ることは不可能
- この研究では以下の手順で抽出したクローンを正解集合として扱った
  - 手順1：Bellonが5人のクローン検出ツール開発者に，8つのソフトウェアに対するクローン検出を依頼
  - 手順2：各開発者は自身のツールを用いてクローンを検出し，検出結果をBellonに送付
  - 手順3：Bellonは，全ての結果から2%をランダムに抽出し，それらがクローンであるかどうかを主観により判断

クローンであると判断されたものが
正解クローンとして扱われた

Bellonらの比較・評価：実験結果（一部）

j2sdk1.4.0-javax.swingに対する結果





- 字句単位は，漏れなく検出できるが誤検出が多い
- メトリクスを用いた検出は，誤検出が比較的少ないが，検出漏れが多い
- …

第三部
CCFinderといくつかの適用事例

クローン検出ツールCCFinder

- ソースコードを字句単位で直接比較することによってクローンを検出
  - ユーザ定義名の置き換え
  - 名前空間の正規化
  - テーブル初期化部分の取り除き
  - モジュールの区切りの認識

- 解析結果はテキスト形式で出力
- 数百万行規模でも実用的な時間で解析可能

ステップ1:字句解析

1. static void foo ( ) throws RESyntaxException {
2. String a [ ] = new String [ ] { "123,400" , "abc" , "orange 100" } ;
3. org . apache . regexp . RE pat = new org . apache . regexp . RE ("[0-9,]+") ;
4. int sum = 0 ;
5. for (int i = 0 ; i < a . length ; ++ i)
6. if (pat . match (a [i]))
7. sum += Sample . parseNumber (pat . getParen (0)) ;
8. System . out . println ("sum = " + sum) ;
9. }
10. static void goo (String a [ ]) throws RESyntaxException {
11. RE exp = new RE ("[0-9,]+") ;
12. int sum = 0 ;
13. for (int i = 0 ; i < a . length ; ++ i)
14. if (exp . match (a [i]))
15. sum += parseNumber (exp . getParen (0)) ;
16. System . out . println ("sum = " + sum) ;
17. }

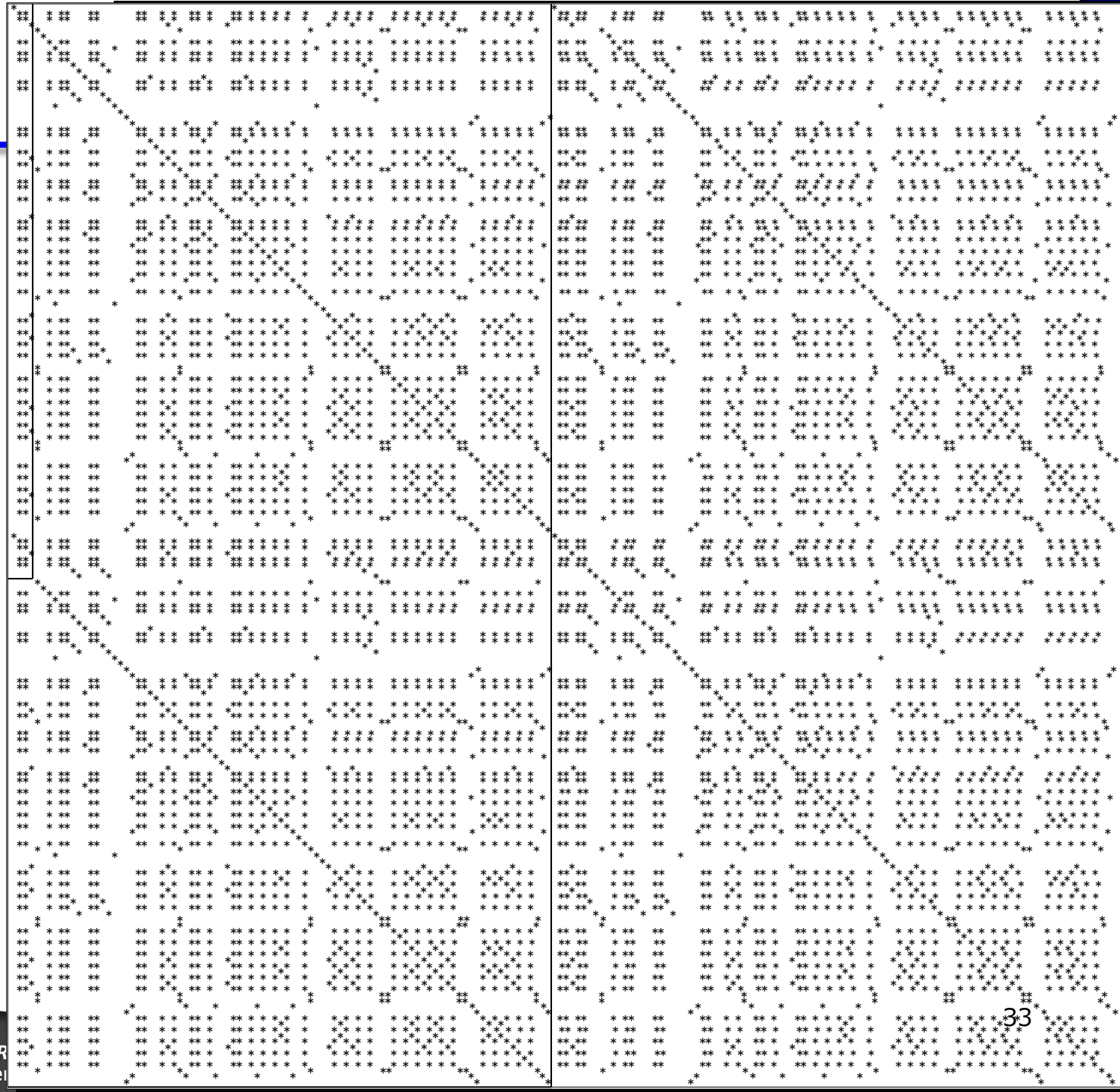
ステップ2：変形ルールの適用

```
1. static void foo ( ) throws RESyntaxException {
2. String a [ ] = new String [ ] { $u } ;
3. RE pat = new RE ( "[0-9,]+" ) ;
4. int sum = 0 ;
5. for ( int i = 0 ; i < a . length ; ++ i )
6. if ( pat . match ( a [ i ] ) )
7. sum += Sample . parseNumber ( pat . getParen ( 0 ) ) ;
8. System . out . println ( "sum = " + sum ) ;
9. }
10. static void goo ( String a [ ] ) throws RESyntaxException {
11. RE exp = new RE ( "[0-9,]+" ) ;
12. int sum = 0 ;
13. for ( int i = 0 ; i < a . length ; ++ i )
14. if ( exp . match ( a [ i ] ) )
15. sum += $p . parseNumber ( exp . getParen ( 0 ) ) ;
16. System . out . println ( "sum = " + sum ) ;
17. }
```


ステップ3:ユーザ定義名の置き換え

```
1. static $p $p ( ) throws $p {
2. $p $p [ ] = new $p [ ] { $u } ;
3. $p $p = new $p ( $p ) ;
4. $p $p = $p ;
5. for ( $p $p = $p ; $p < $p . $p ; ++ $p )
6. if ( $p . $p ( $p [ $p ] ) )
7. $p += $p . $p ( $p . $p ( $p ) ) ;
8. $p . $p . $p ( $p + $p ) ;
9. }
10. static $p $p ( $p $p [ ] ) throws $p {
11. $p $p = new $p ( $p ) ;
12. $p $p = $p ;
13. for ( $p $p = $p ; $p < $p . $p ; ++ $p )
14. if ( $p . $p ( $p [ $p ] ) )
15. $p += $p . $p ( $p . $p ( $p ) ) ;
16. $p . $p . $p ( $p + $p ) ;
17. }
```

ステップ4:
マッチングアルゴリズムを利用してクローンを検出



ステップ5：ソースコードへのマッピング

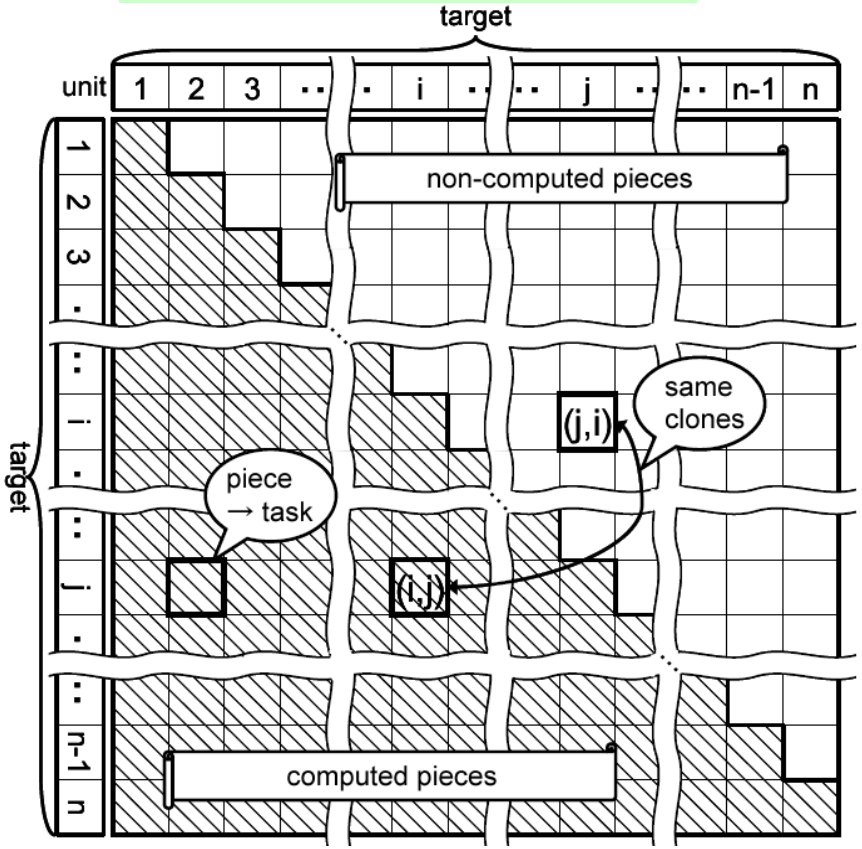

```
1.  static void foo() throws RESyntaxException {
2.    String a[] = new String [] { "123,400", "abc", "orange 100" };
3.    org.apache.regexp.RE pat = new org.apache.regexp.RE("[0-9,]+");
4.    int sum = 0;
5.    for (int i = 0; i < a.length; ++i)
6.      if (pat.match(a[i]))
7.        sum += Sample.parseNumber(pat.getParen(0));
8.    System.out.println("sum = " + sum);
9.  }
10.  static void goo(String [] a) throws RESyntaxException {
11.    RE exp = new RE("[0-9,]+");
12.    int sum = 0;
13.    for (int i = 0; i < a.length; ++i)
14.      if (exp.match(a[i]))
15.        sum += parseNumber(exp.getParen(0));
16.    System.out.println("sum = " + sum);
17.  }
```

CCFinderを利用した主な活動

- ソフトウェア工学工房
  - 阪大が作成したツールの紹介，分析法の解説
  - 企業からの適応事例の報告
  - http://sel.ist.osaka-u.ac.jp/kobo/
- CCFinder/ICCAパッケージの配布
  - 国内外の組織・個人への提供（200～300）
- 多数の企業のソフトウェアへの適用・共同研究
- 後継機CCFinderX
  - http://www.ccfinder.net/ccfinderx-j.html

分散環境を利用した超大規模クローン検出[3]

各格子がクローン検出の単位



大阪大学基礎工学部情報科学科演習室の80台のワークステーションを利用

[3] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue, “Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder”, Proc of the 29the IEEE International Conference on Software Engineering (ICSE2007), pp.106–115, May 23–25, 2007.

COSEプロジェクトへの適用：概要

- COSEプロジェクトで実施されたソフトウェアに対して行った，クローン分析による理解支援
  - 実装は5社が担当
  - http://www.empirical.jp/download/past/publicdata/15th_kenkyukai/2_katsumata.pdf
- 単体テスト後のコードと結合テスト後のコードについて分析
- 制約下での分析
  - 5社のソースコードの分析時間は約6時間
  - 分析者はソースコードに関する知識がない

第四部
コードクローン検出手法の例

コードクローン検出手法の例

- プログラム依存グラフ(PDG)を用いた検出手法の改良[4]
  - 情報処理学会 50周年記念論文賞
  - 情報処理学会 コンピュータサイエンス領域奨励賞

- 概要
  - PDGを用いた検出手法の弱点を和らげる手法を2つ提案し，検出速度および検出能力が向上したことを実験により確認

[4] 肥後芳樹, 楠本真二, “プログラム依存グラフを用いたコードクローン検出法の改良と評価”, 情報処理学会論文誌, Vol.51, No.12, pp.2149–2168, 2010年12月.

PDGを用いた検出

- PDG上の同形部分グラフがクローンとして検出される
- 長所
  - TYPE-3クローンを検出できる
- 短所
  - 行単位や字句単位，抽象構文木を用いた手法に比べ，TYPE-1およびTYPE-2クローンの検出能力が弱い
  - 検出に必要な計算コストが高く，実規模のソフトウェアに対しては適用が難しい

新しい依存辺を導入

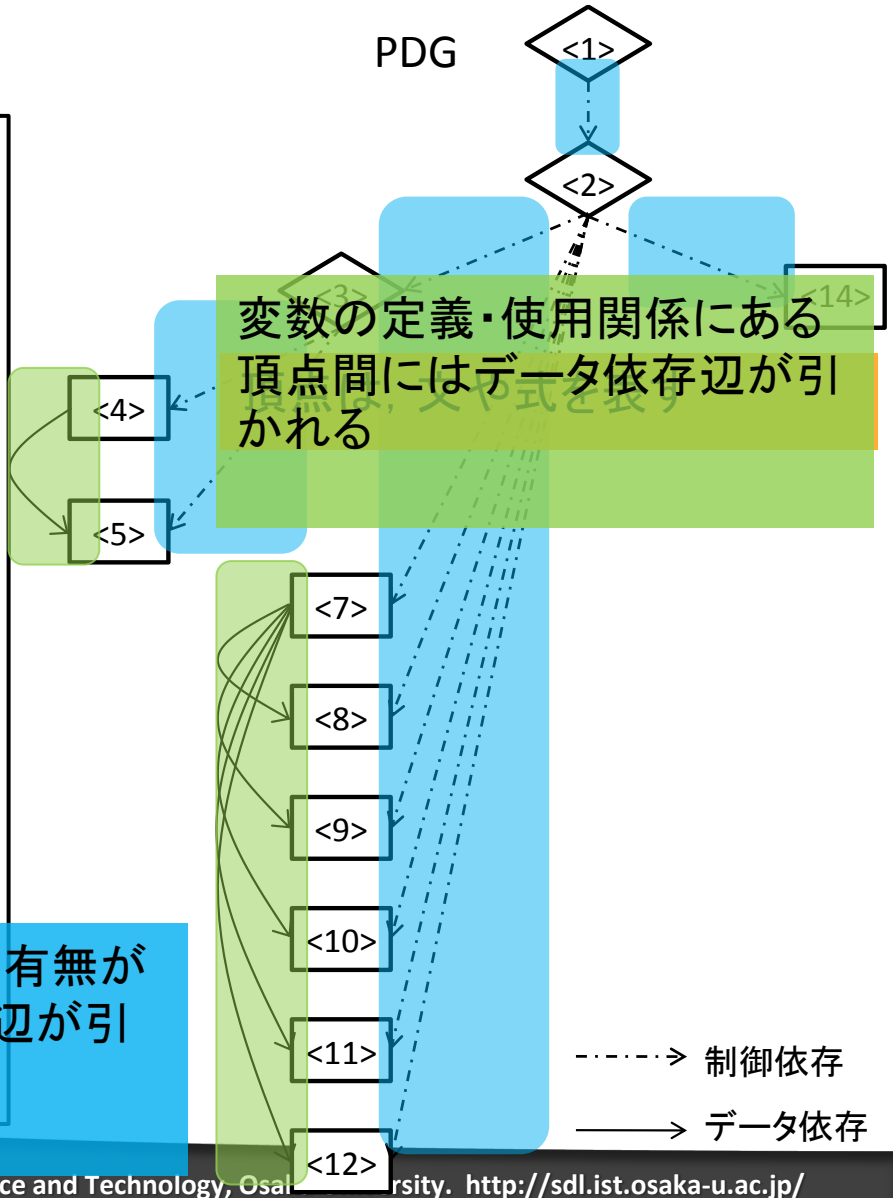
頂点数の削減

ソースコードとPDGの例

ソースコード

```
 1: String sample1(){
 2:   if(this.trueOrFalse()){
 3:    if(null == this.getPath()){
 4:     Project proj = this.getProject();
 5:     this.setPath(proj.getBaseDir());
 6:    }
 7:    StringBuilder text =
                      new StringBuilder();
 8:    text.append( “String A” );
 9:    text.append( “String B” );
10:    text.append( “String C” );
11:    text.append( “String D” );
12:    return text.toString();
13:   }else{
14:    return "";
15:   }
16: }
```

PDG



変数の定義・使用関係にある頂点間にはデータ依存辺が引かれる

条件式と，その結果により実行の有無が決定される文の間には制御依存辺が引かれる

PDGを用いたクローンの検出手順

- 手順１：PDGの全ての頂点のハッシュ値を求めて，ハッシュ値が等しい頂点ごとにグループを作成
- 手順２：同じグループ内の全ての頂点のペアから，同形部分グラフのペアの探索
- 手順３：不必要なクローンのペアを削除
- 手順４：クローンのペアからクローンのセット（同値類）の作成

手順３と手順４の説明は割愛

手順1：ハッシュ値の生成

- PDGの全ての頂点（プログラム中の文や式）のハッシュ値を求めて，ハッシュ値が同じ頂点毎にグループを作成する
  - ハッシュ値は，頂点が表すプログラム要素の構造にもとづいて計算される
  - ハッシュ値の計算前に，変数やリテラルはその「型」に変換される
- 構造が等しく，用いている変数やリテラルの型が同じであれば，同じハッシュ値が生成される
  - 同じハッシュ値：`int i = 0, int number = 0`
  - 違うハッシュ値：`int x = n, int y = m + z.methodZ()`

ハッシュ値の作成例（ソースコード）

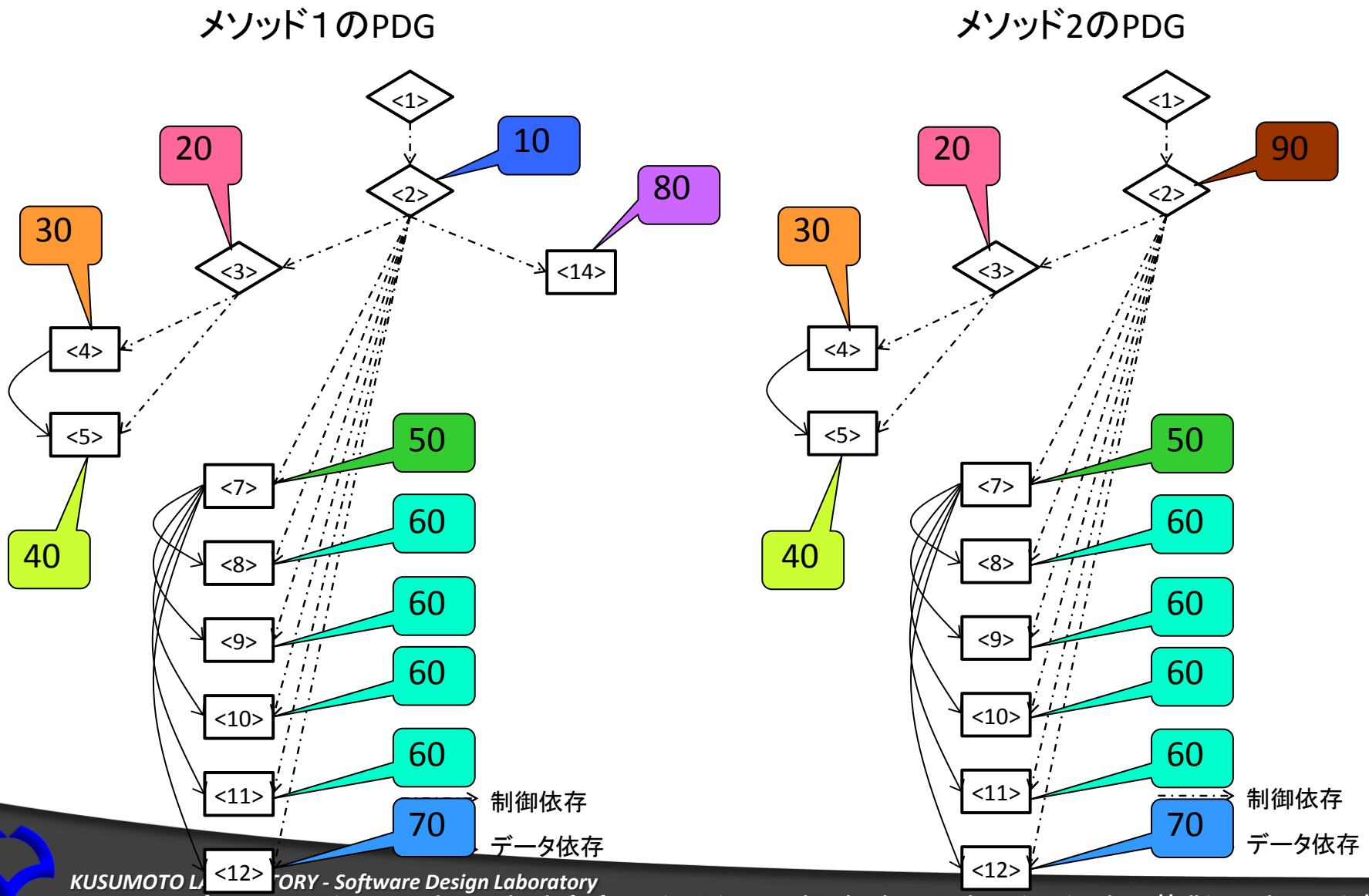
メソッド1

```
 1: String sample1(){
 2:  if(this.trueOrFalse()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                    new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }else{
14:   return “”;
15:  }
16: }
```

メソッド2

```
 1: String sample2(){
 2:  while(this.a > this.b()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                    new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }
14: }
```

ハッシュ値の作成例（PDG）

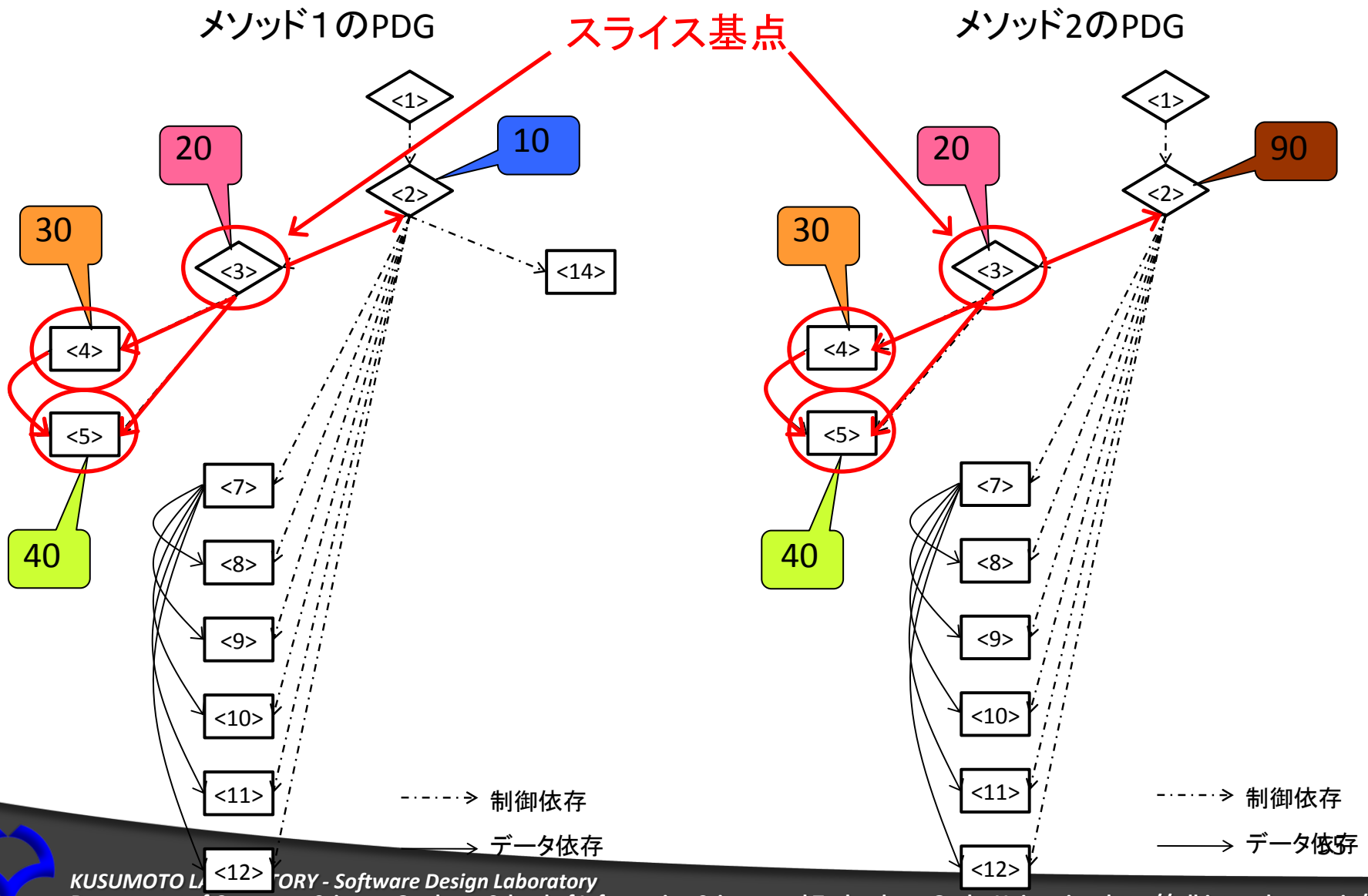


手順2：同形部分グラフの検出

- 同じグループに属する頂点のペア（r1, r2）から，同形部分グラフを検出する
  - r1とr2を起点とするスライシングにより，新たにたどった頂点が同じハッシュ値を保つ場合は，同形部分グラフに加える

- 以下の条件を満たすとき，スライシングが終了する
  - たどった頂点のペア（p1, p2）が異なるハッシュ値を持つ
  - ハッシュ値は等しいが，r1のグラフに既にp1が含まれている
    - 無限ループを回避するための処理
  - ハッシュ値は等しいが，r1のグラフに既にp2が含まれている
    - 2つの同形部分グラフが頂点を共有するのを回避するための処理

同形部分グラフ検出例


メソッド1のPDG
スライス基点
メソッド2のPDG
<1>
<2>
<3>
<4>
<5>
<7>
<8>
<9>
<10>
<11>
<12>
<14>
10
20
30
40
90
制御依存
データ依存

他の検出手法を用いた場合は…

メソッド1

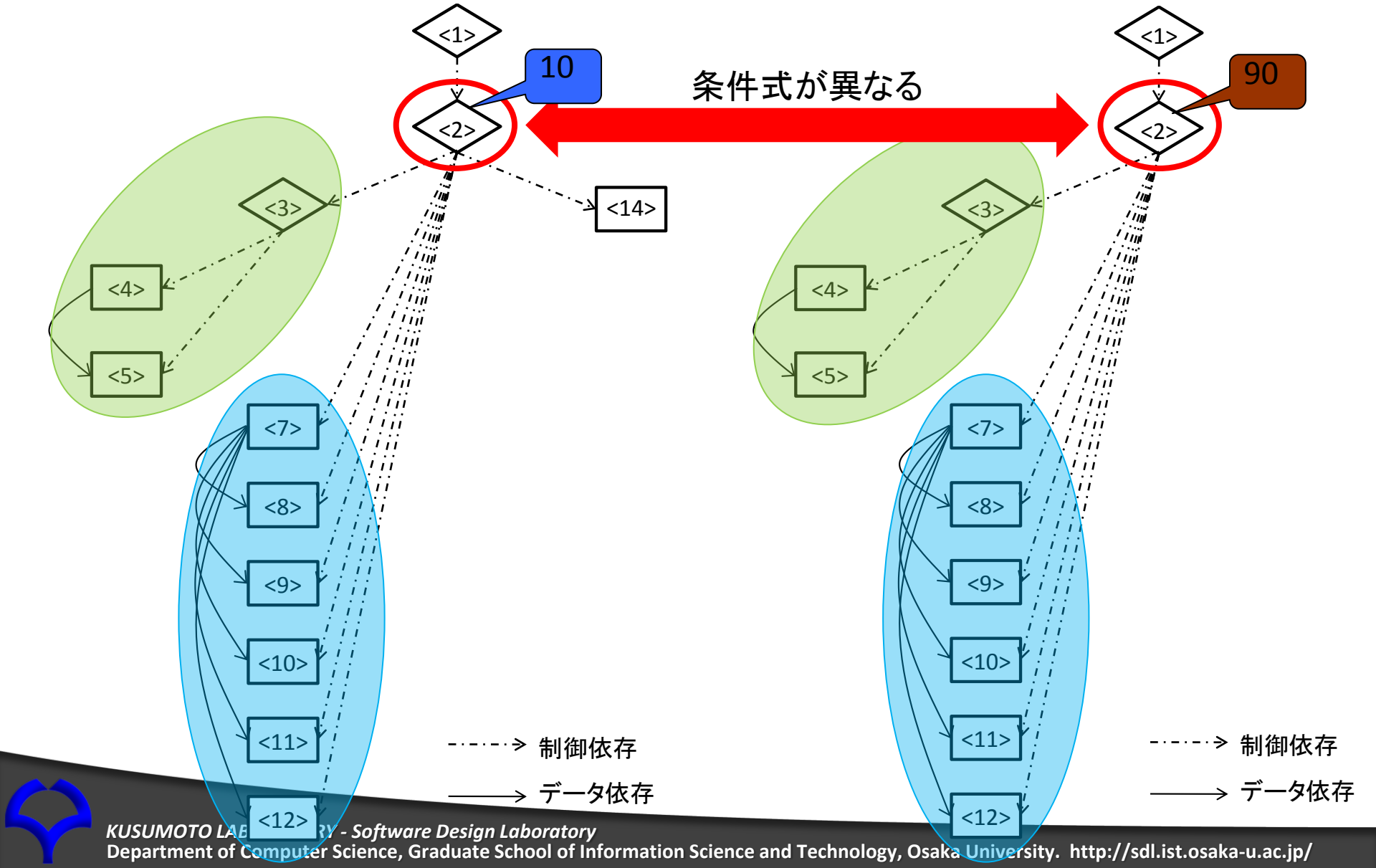
```
 1: String sample1(){
 2:  if(this.trueOrFalse()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                   new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }else{
14:   return “”;
15:  }
16: }
```

メソッド2

```
 1: String sample2(){
 2:  while(this.a > this.b()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                   new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }
14: }
```

行単位や軸単位の検出手法を用いた場合は，赤色の部分（　）の部分がクローンとして検出される

PDGを用いた検出では<2>でスライシングが止まってしまう



PDGを用いた検出では〈2〉でスライシングが止まってしまう

メソッド1

```
 1: String sample1(){
 2:  if(this.trueOrFalse()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                   new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }else{
14:   return “”;
15:  }
16: }
```

メソッド2

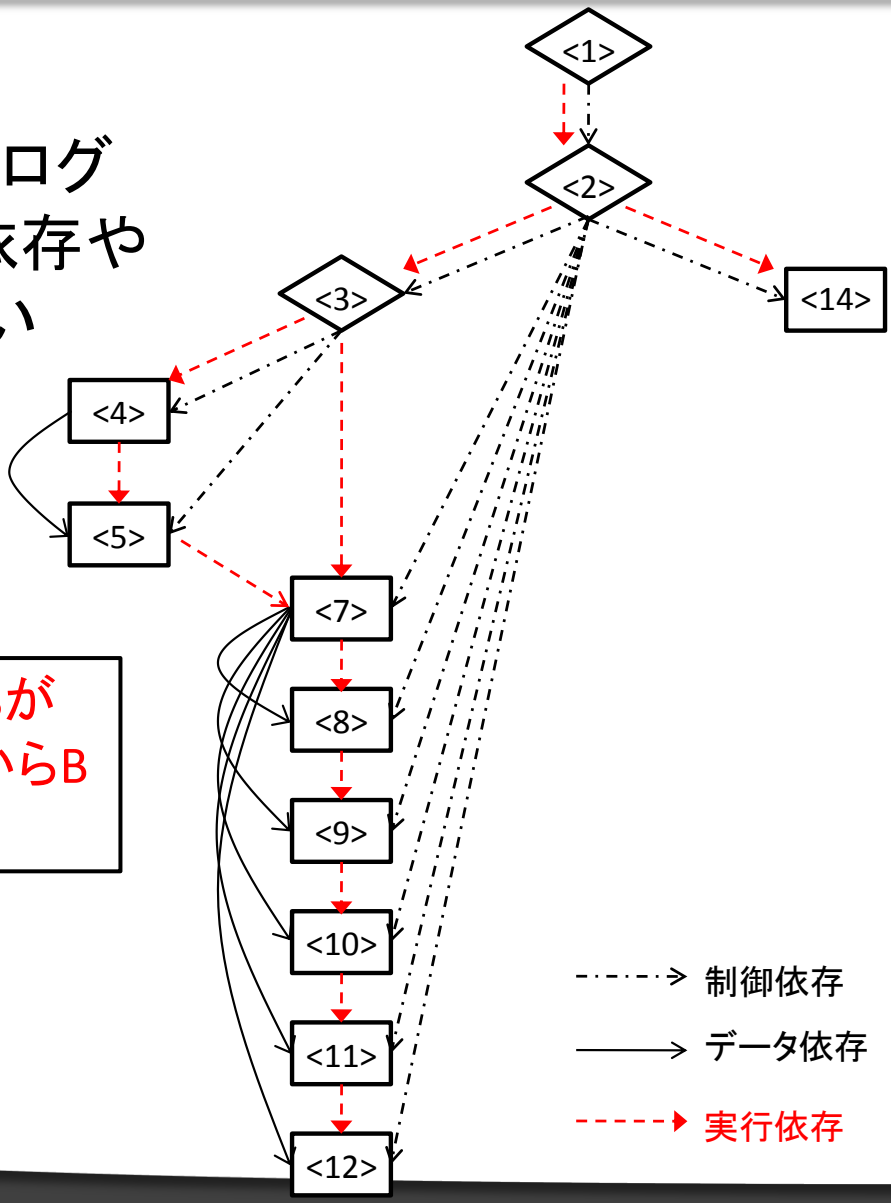
```
 1: String sample2(){
 2:  while(this.a > this.b()){
 3:   if(null == this.getPath()){
 4:    Project proj = this.getProject();
 5:    this.setPath(proj.getBaseDir());
 6:   }
 7:   StringBuilder text =
                   new StringBuilder();
 8:   text.append(“String A”);
 9:   text.append(“String B”);
10:   text.append(“String C”);
11:   text.append(“String D”);
12:   return text.toString();
13:  }
14: }
```

従来のPDGを用いた場合は，2つの異なるコードクローンとして検出されてしまう

提案手法:検出能力の改善

- 検出能力が低い原因
  - ソースコード上の隣接したプログラム要素が必ずしもデータ依存や制御依存をもつわけではない
- 解決策
  - 実行依存辺の導入

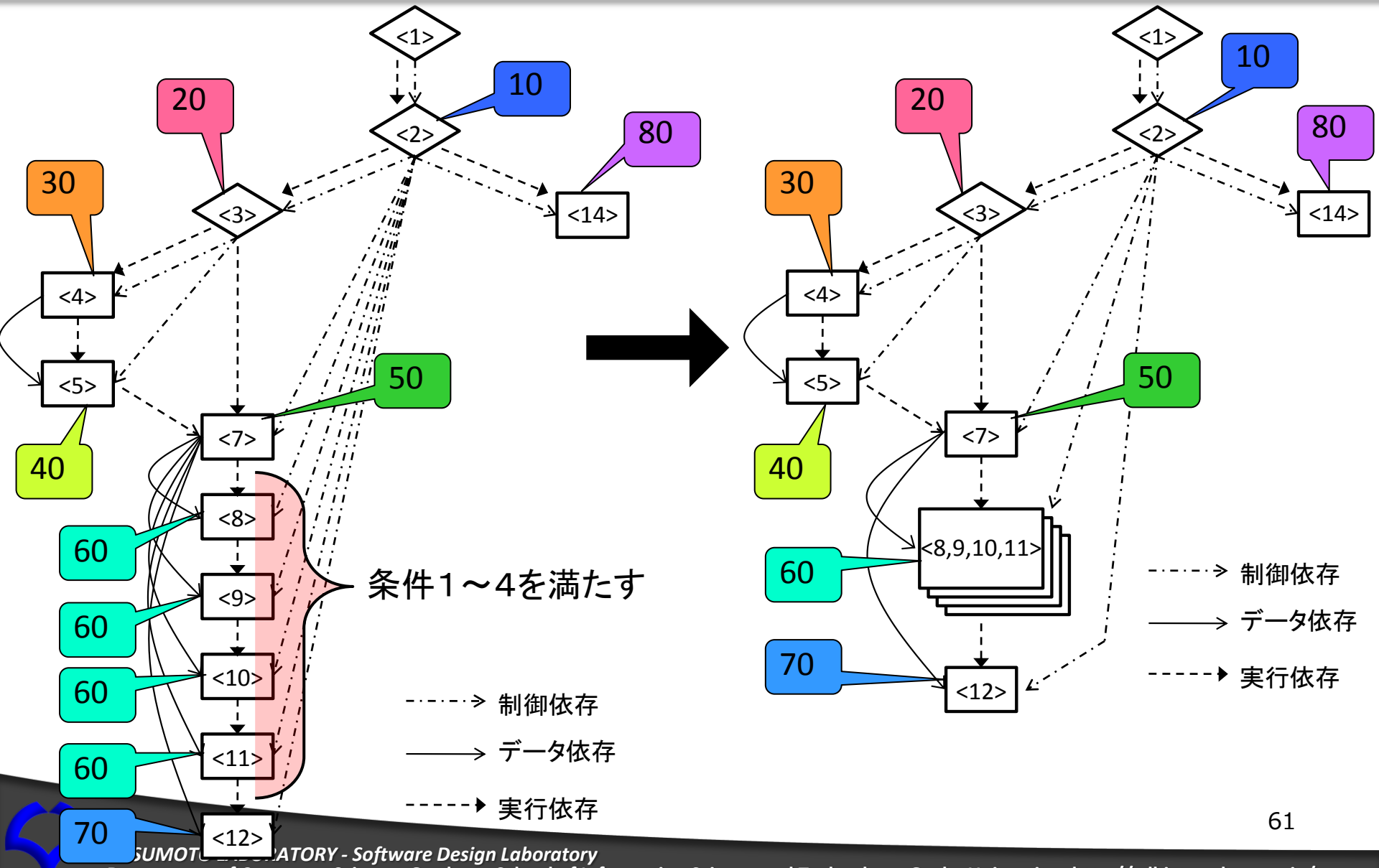
頂点Aが実行された直後に，頂点Bが実行される可能性がある場合，AからBに対して実行依存辺が引かれる



提案手法：計算コストの削減手法

- PDG上の頂点を集約（複数の頂点を1つにまとめる）することにより，計算コストの削減を図る
- 以下の条件を満たす頂点群が1つの頂点に集約される
  - 条件1：要素sから要素tへ，実行依存辺のみをたどることにより到達可能な経路（以降，経路R）が存在する
  - 条件2：経路Rに存在する全ての頂点は，実行依存の入力辺と出力辺を1つずつ持つ
    - if文やwhile文などの条件式，およびそれらが終わった後の合流点を含まない
  - 条件3：経路R上に存在する全ての頂点は，同一ハッシュ値を持つ
  - 条件4：経路Rを包含するいかなる経路も，条件2と条件3を同時に満たさない

例題のPDGに適用すると．．．



計算コスト削減手法の効果

- 集約した頂点がスライス上に現れる場合

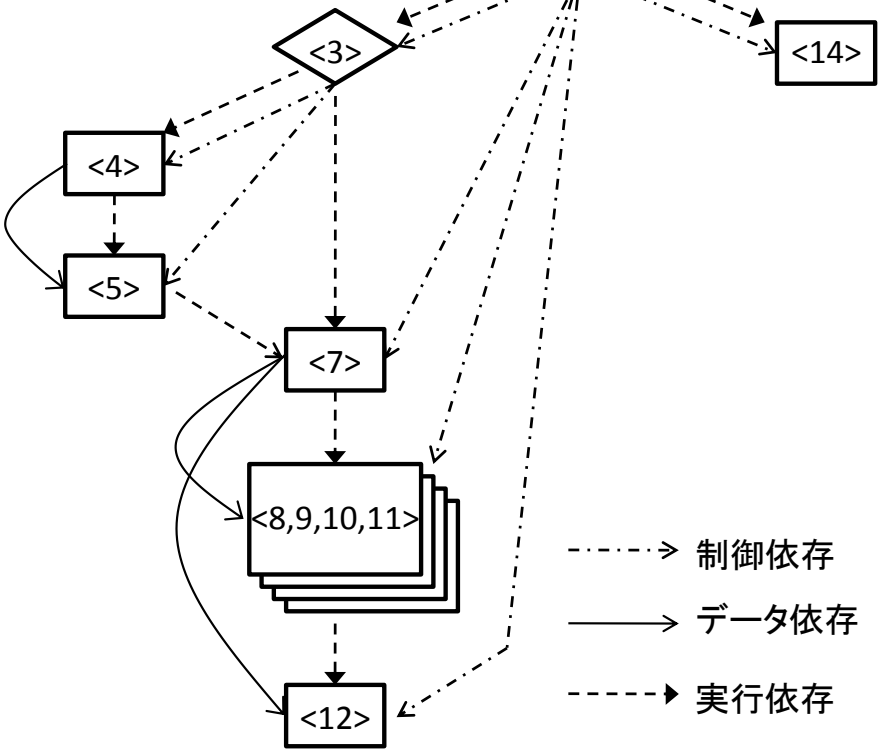
  頂点をたどるコストを抑えれる

- 集約した頂点がスライス基点となる場合

  スライシングの数を削減できる

<8>から<11>の頂点のグループがスライス基点となる場合
集約有り：$_2C_2$=1個のペア
集約なし：$_8C_2$=28個のペア

<7>から<12>にたどる場合
集約有り：2ホップ
集約なし：5ホップ



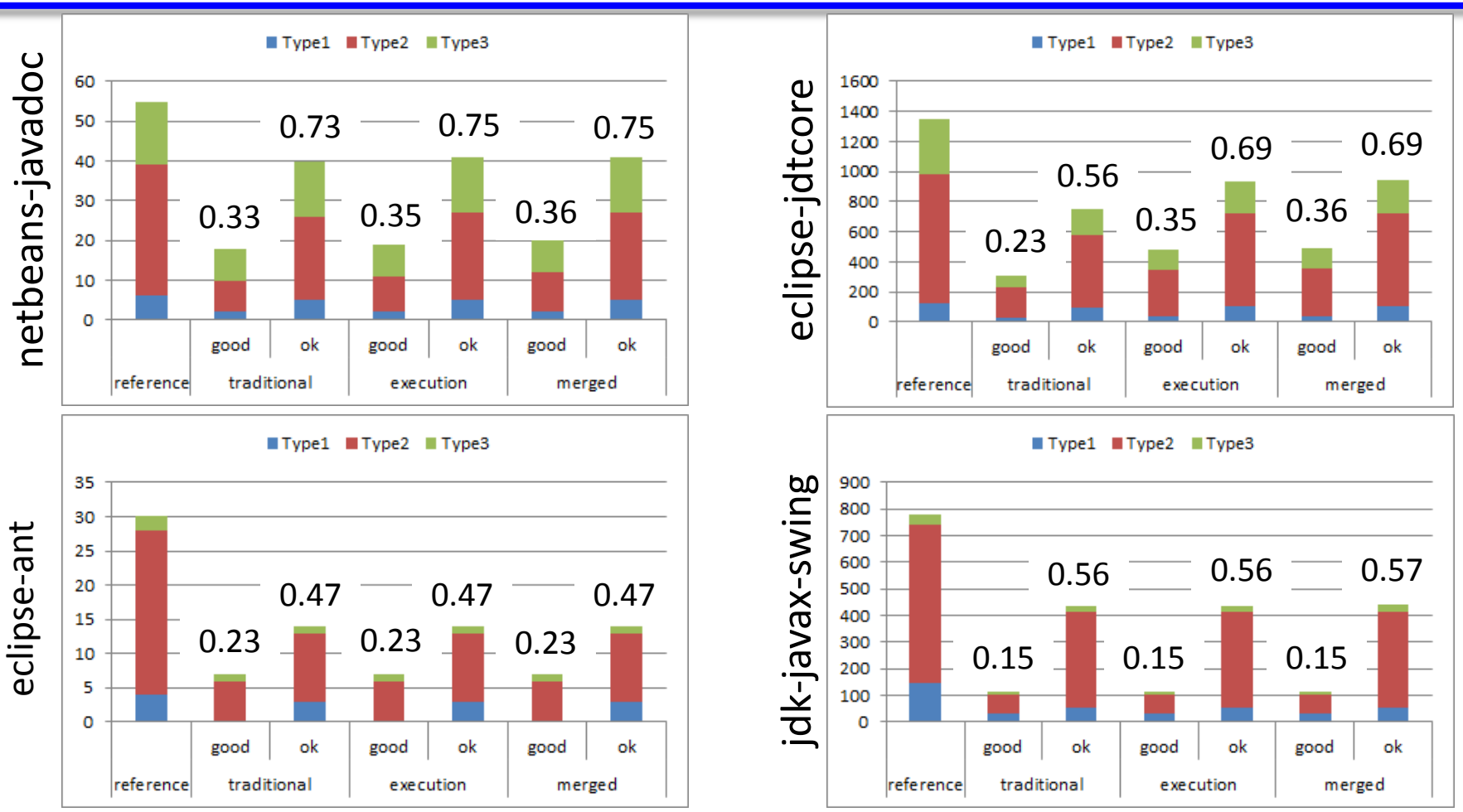
計算コストの評価

traditional を基準とした，
各PDGの頂点比較回数の割合

ソフトウェア	PDG	頂点数	辺数			頂点比較回数	
			データ依存	制御依存	実行依存	絶対値	増減率
netbeans	traditional	6,557	4,700	5,626	0	110,422,894	100.0%
	execution	6,557	4,700	5,626	6,144	103,408,483	93.6%
	merged	6,060	4,362	5,131	5,647	40,211,133	36.4%
ant	traditional	12,505	11,269	10,423	0	10,437,444	100.0%
	execution	12,505	11,269	10,423	12,379	11,396,350	109.2%
	merged	12,126	10,998	10,073	12,002	9,916,301	95.0%
jdtcore	traditional	77,493	91,617	64,701	0	1,424,793,100	100.0%
	execution	77,493	91,617	64,701	77,980	1,480,543,317	103.9%
	merged	73,885	88,443	61,263	74,595	1,048,732,585	73.6%
swing	traditional	82,824	75,560	68,310	0	528,247,797	100.0%
	execution	82,824	75,550	68,310	78,110	533,917,155	101.1%
	merged	78,783	73,026	64,370	74,050	409,070,104	77.4%

- execution の増減率は93～109%（3つのソフトウェアで増加）
- merged の増減率は38～95%（全てのソフトウェアで減少）
  - 頂点数や辺数は数%しか減少していないことを考慮すると，検出コストのボトルネック部分をうまく集約できているといえる

検出結果(再現率)の評価



- traditionalやexecutionでは検出されるが，mergedでは検出されない正解クローンは皆無
- mergedを用いることにより新しく見つかる正解クローンが存在

第五部
同じ機能を持つコードを見つけるための調査

同じ機能を持つコードを見つけるための調査

- 構造の類似度を用いるだけでは，同じ機能を持つコードを見つけるのに十分ではないことを示した[5]
  - FSE2014に採録(61/273≒22%)

- 概要
  - 構造の類似度，語彙の類似度，メソッド名の一致・不一致の3つを利用して，同じ機能を持つコードを見つけれるか，見る価値が無いコードを見つけてしまうかを手作業により調査

[5] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on Java methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2014).

調査の背景

- ソースコードの構造の類似度や語彙の類似度を利用した研究が多く行われている
  - 構造の類似度：クローン検出
  - 語彙の類似度：キーワードを用いたコード検索

- 既存研究は以下の前提の上で行われている
  - クローン検出：構造が類似していれば，その部分の機能も似ているはず
  - コード検索：語彙が類似していれば，その部分の機能も似ているはず

これらの前提はいつも成り立つわけではない

前提が成り立たない例

- クイックソートとバブルソート
  - 語彙は類似しているはず
    - sort, array, …
  - 異なるアルゴリズムのため構造は類似していない

- C/C++やJavaのfor文
  - 構造は類似している
    - 配列やリストの各要素に対して処理を行う
  - 語彙が類似しているとは限らない
    - どのような場面でfor文が利用されているのかに依存

さらに…

- コードがdeclarative unitな場合，それらは名前を持つ
  - Javaであれば，クラスやメソッド
- Javaにおいてメソッドのシグネチャは最も情報量が多い[3]
  - mainやactionPerformed等，メソッド名がその機能を表していない場合も多々ある

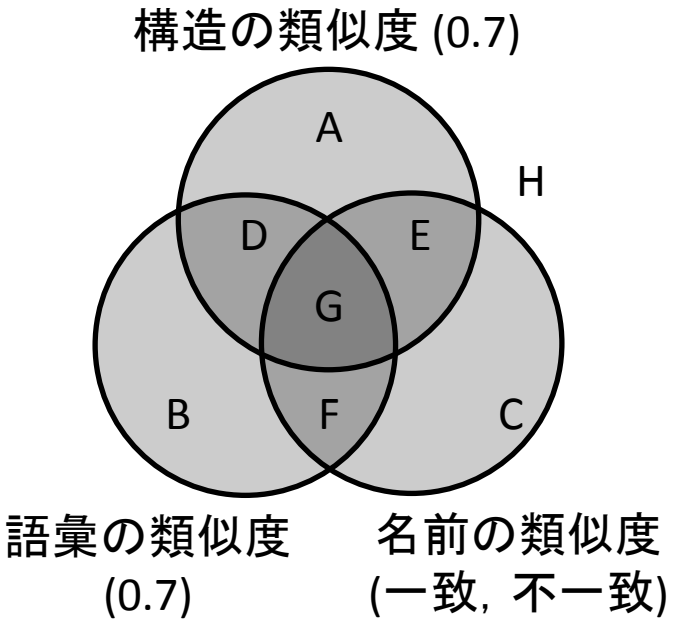
[3] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the Use of Lexical Information for Software System Clustering. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR '11), 35-44.

この調査の概要

- 目的
  - 構造類似度，語彙類似度，名前類似度をどのように利用すれば，同じ機能を持つコードを見つけれるかの知見を得ること
- 方法
  - 3つの類似度と同じ機能を持つコードの関係を手作業により調査
- 対象
  - 約1400万のJavaメソッドペア群（2つ）

3つの類似度から得られる8つの領域

- 各メソッドペアはいずれかの領域に含まれる



領域	APACHE	UCI
A	45	161
B	149	355
C	29,591	37,047
D	229	598
E	82	80
F	98	176
G	472	1,918
~~H~~	~~14,709,069~~	~~14,873,636~~

- 各領域のメソッドペアを手作業により調査
  - 100以下のメソッドペアを持つ領域は，全てのペアを調査
  - 100より多いメソッドペアを持つ領域は，100のペアを調査

メソッドペアのカテゴリ

WF（Within File）：同じファイル内にあるメソッドのペア

WD（Within Directory）：異なるファイルだが同じディレクトリにあるメソッドのペア

WP（Within Project）：異なるディレクトリだが，同じプロジェクト内にあるメソッドのペア

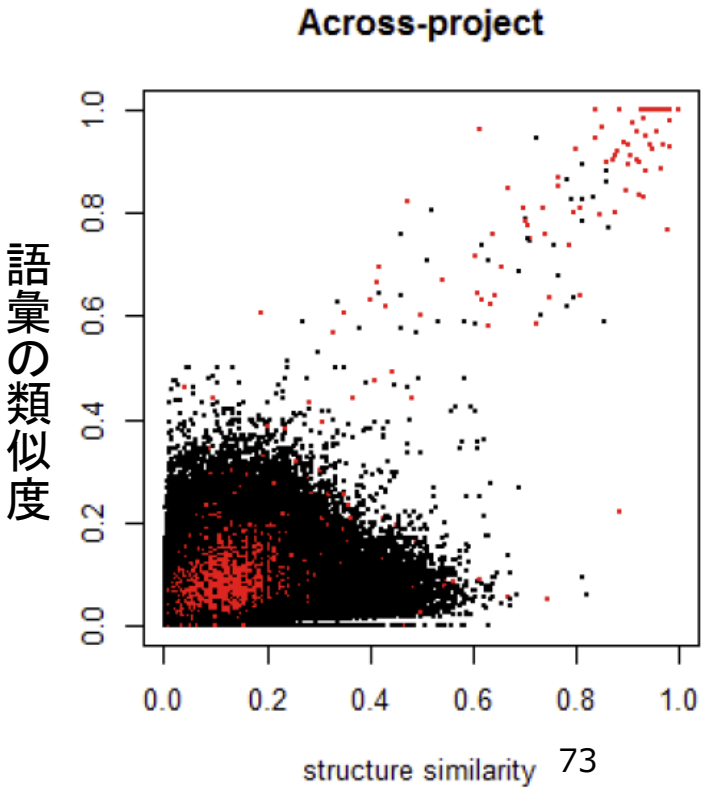
AP（Across Project）：異なるプロジェクトにあるメソッドのペア

領域C

構造の類似度：似てない
語彙の類似度：似てない
名前の類似度：一致

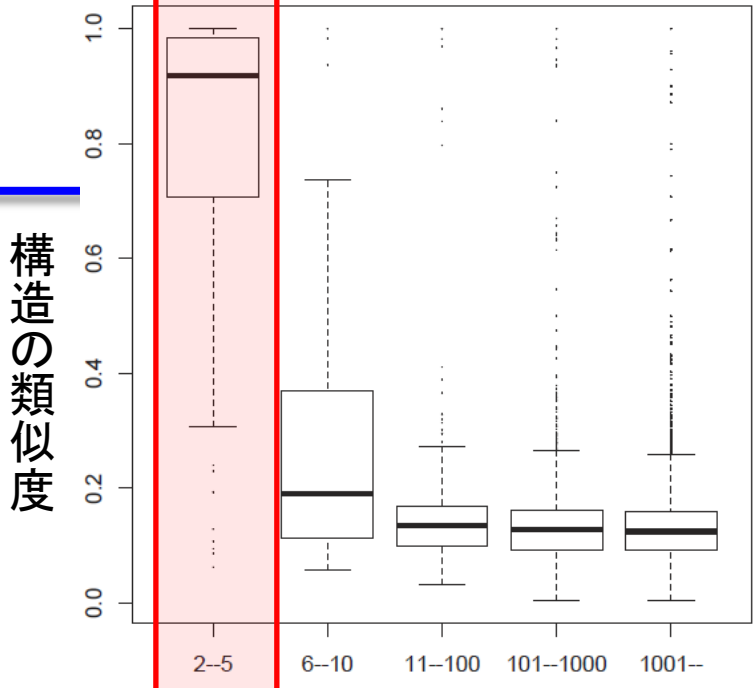
- 調査したペアは，関係なさそうなメソッドばかり
  - 見つける価値なし
  - getやexecute等の抽象度が高い名前，mainやaddActionListenerなどの言語に依存した名前

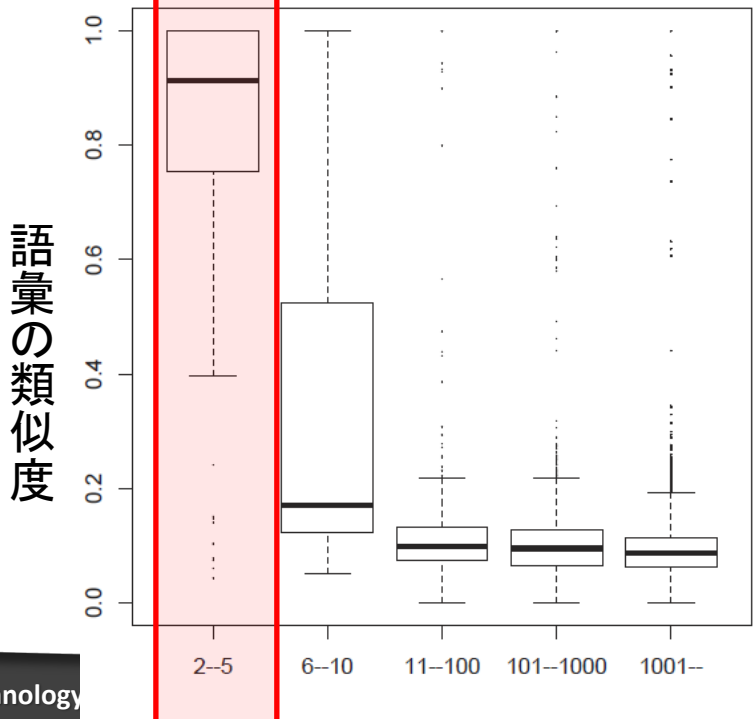
- 同じ名前を持つメソッドのペア（赤の点）は左下に密集している
  - しかし一部のペアは左上に存在
  - なにが違うのか？



領域C

- 横軸は同じ名前を持つメソッドの数
  - 一番左の棒グラフは，同じ名前を持つメソッドが5つ以下の場合の構造／語彙の類似度の分布
- 同じ名前を持つメソッドの数が少ないほど，それらの間の構造／語彙の類似度は高い

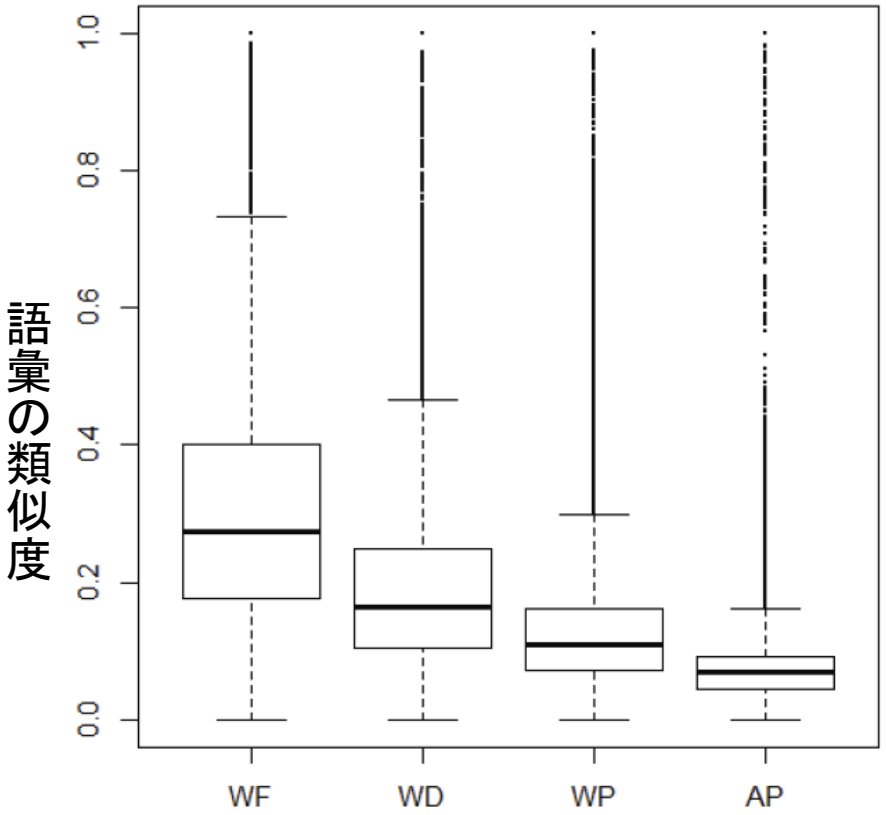




領域Aと領域E

構造の類似度：似ている
語彙の類似度：似てない
名前の類似度：一致，不一致

- カテゴリAPでは，多くのメソッドペアが言語依存の実装
  - 連続したcase文やif-else文
- カテゴリWPやWDは，同じ処理を異なる型に対して行っていた
- カテゴリWFは，たった14のメソッドペアしか存在しなかった



領域Bと領域F

構造の類似度：似てない
語彙の類似度：似ている
名前の類似度：一致，不一致

- カテゴリAP
  - プロジェクト間のコード再利用を多数発見
    - コピペしたあとに大きく修正しているため，構造の類似度は高くない
    - しかし，語彙はそのまま利用されているため，語彙の類似度は高い
- カテゴリWPとWD
  - メソッドの一部分がクローンになっている
  - 関連するメソッド（compressとuncompress，ANDとORの計算）
- カテゴリWF
  - privateフィールドやメソッドを共有しているメソッド

領域Dと領域G

構造の類似度：似ている
語彙の類似度：似ている
名前の類似度：一致，不一致

- 関係ないメソッドがペアになっているものは皆無
  - 構造の類似度と語彙の類似度が100%の時は，メソッド名は常に一致
- カテゴリAPは，プロジェクト間のコード再利用
- カテゴリWPとWDは，リファクタリングの候補となりそうなメソッドのペア

括弧内の数値は，構造の類似度と語彙の類似度がどちらも100%の場合

	Within-directory	Within-project
(A) 領域Dと領域Gのペア数	124	257
(B) 共通の親クラスを持つ	43 (5)	47 (19)
(A)に対する(B)の割合	0.34 (0.04)	0.18 (0.07)

得られた知見

- 少数のメソッドが同じ名前を保つ場合，その名前には意味がある（類似機能を持つ）
- 構造の類似度を用いた場合（クローン検出）では，コードの再利用を見逃す場合が多い
- 構造の類似度で見つけてしまう誤検出は，語彙の類似度を用いることで防げる
- 語彙の類似度は，ファイル内メソッドペアの場合には役に立たない

第六部
これからに向けて

(私が思う)最近のトレンド

- 保守への悪影響度の調査
  - クローンにはバグが含まれるか？
  - 同時修正や遅延伝播は起こっているのか？
- 集約から管理へ
  - クローンを無くすのではなく，クローンをうまく付き合っていく
  - 可視化，漏れのない修正支援
- 超大規模ソースからのクローン検出
  - より速く，より細粒度で
- 他の研究の要素技術としての利用
  - リポジトリマイニング，再利用

(私が思う)これから必要な研究

- 検出手法の比較・評価方法
  - 正解クローン作成に主観が(極力)入らないように
  - 多量の正解クローンが生成可能
- 現状で検出できていないクローンを検出できる手法
  - クローン検出技術はクローン研究の根幹
  - 構造以外の情報を使う?
- 誤検出を無くす・少なくする後処理手法
  - クローン技術に失望する人の多くは，誤検出に辟易したことが原因
- 商用ソフトウェアへの適用
  - これまでの研究の多くはオープンソースへの適用
- 限定的なコンテキストで超便利な手法
  - 色んな場面で使える手法よりは，1つの場面でしか使えなくてもすごく有用な手法が良い

研究を行う心構え：手を動かす

論文を読むだけでは既存手法の細部まではわからない，
興味があるものを見つけて実際に動かすことが大事

百聞は一見に如かず，百見は一試に如かず

紹介したPDGを用いた手法は，なにもアイディアがない状態で，既存手法を実装することから始めた

- 実際にPDGで検出されたクローンを確認することにより，既存手法の問題点を正確に認識することができる
- 正確な認識は，どのような改良が必要かのアイディアにつながる

研究を行う心構え：あきらめない

査読結果で悲観的にならない，不採録だった場合には
自分の研究をより良くするチャンスができたと考える

為せば成る，為さねば成らぬ何事も，成らぬは人の為さぬなりけり

紹介した同じ機能をもつコードを見つけるための調査は，当初はICPC2013に投稿したが，不採録だった

- 全ての査読コメントに対応できるようにやり直し，FSE2014に採録された
- ICPC2013の不採録通知（2013年４月）からFSE2014採録（2014年3月）までは約１１ヶ月

研究を行う心構え：巨匠に学ぶ

- 大阪大学 井上先生
  - ソフトウェアに関する論文の書き方
  - http://sel.ist.osaka-u.ac.jp/lab-db/betuzuri/archive/746/746.pdf
- 九州大学 鵜林先生
  - 世界を目指す論文の書き方～不採録コメントに学ぶ～
  - http://ws.cs.kobe-u.ac.jp/~masa-n/ses2011/ses2011_tutorial2.pdf
- 大阪大学 原先生
  - 一流論文誌・国際会議に採択されるための研究「心・技・体」
  - http://www.ipsj.or.jp/journal/info/hara75.pdf

コラボレーションしましょう

- クローンに興味があれば話しかけてください
  - 企業・大学どちらも大歓迎
  - win−winになれるように

SANER 2016

23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering

ABOUT SANER

SANER is the premier research conference on the theory and practice of recovering information from existing software and systems. It explores innovative methods of extracting the many kinds of information that can be recovered from software, software engineering documents, and systems artifacts, and examines innovative ways of using this information in system renovation and program understanding.

SANER promotes discussion and interaction among researchers and practitioners about the development of maintainable systems, and the improvement, evolution, migration, and reengineering of existing systems. It also explores innovative methods of extracting the many kinds of information of interest to software developers and examines innovative ways of using this information in system renovation and program understanding.

SANER will feature technical research paper sessions, workshops, tutorials, an early research achievements track, an industry paper track, a tool demonstration track, a doctoral symposium, and a special track on international research projects within the field of software maintenance and reengineering.

IEEESANER @SANERconf 14 Aug
The #SANER16 research track submission deadline is in 3 months (Nov. 13). Perfect time to start investigating your craziest research ideas !
Expand

IEEESANER @SANERconf 17 May
Less than 6 months before the #SANER16 research track

参考資料

good値とok値（1/2）

- 2つのクローンペアの一致度を表す指標
  - 1つを正解クローン，他方を検出されたコードクローンとして値を算出．検出されたコードクローンが正解クローンとして良いかどうかを判断する
  - 閾値として，0.7を用いる
- 定義１：2つのコード片（f_1とf_2）の重なりの程度を以下の式で定義する．なお，*lines(f)*はコード片fに含まれる行の集合を表す

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|}$$

- 定義２：あるコード片（f_1）が他のコード片（f_2）に含まれている程度を以下の式で定義する

$$contain(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|}$$

good値とok値（2/2）

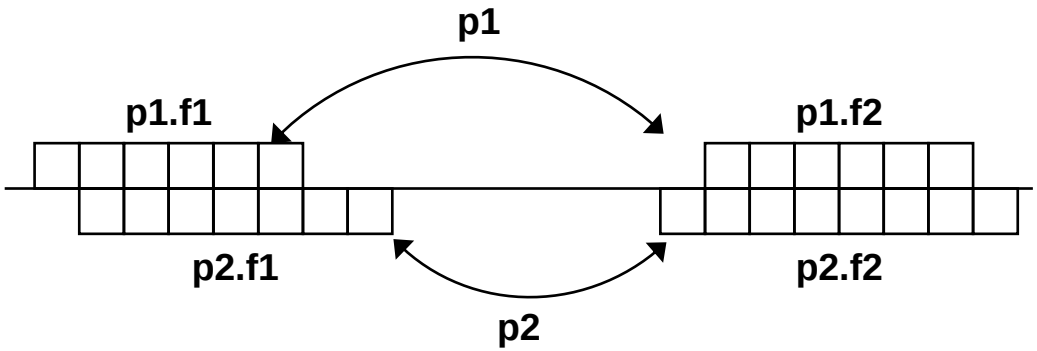
- 定義3：2つのクローンペア（p_1とp_2）のgood値は以下の式で定義される

$$good(p_1, p_2) = \min(overlap(p_1.f_1, p_2.f_1), overlap(p_1.f_2, p_2.f_2))$$

- 定義4：2つのクローンペア（p_1とp_2）のok値は以下の式で定義される

$$ok(p_1, p_2) = \min(\max(contain(p_1.f_1, p_2.f_1), contain(p_2.f_1, p_1.f_1)), \max(contain(p_1.f_2, p_2.f_2), contain(p_2.f_2, p_1.f_2)))$$

good値とok値の計算例



Clone Pair: **p1**, **p2**
Code Fragments in **p1**: **p1.f1**, **p1.f2**
Code Fragments in **p2**: **p2.f1**, **p2.f2**

- good値は，

$$good(p_1, p_2) = \min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8} < 0.7$$

となり，閾値が0.7の場合，p_1とp_2は一致しないと判定

- ok値は，

$$ok(p, p) = \min(\max(\frac{5}{6}, \frac{5}{7}), \max(\frac{6}{6}, \frac{6}{8})) = \frac{5}{6} > 0.7$$

となり，閾値が0.7の場合，p_1とp_2は一致すると判定

Dataset

- Two datasets
  - APACHE: the entire set of Java projects in the *Apache Software Foundation (a snapshot at 2013/Oct/31)*
  - UCI: a huge set of Java software projects (13,000 projects). We randomly selected 500 projects from it
- Data cleansing
  - removing *branches* and *tags* directories
  - removing generated and test files
  - removing small methods
    - Including 50 or fewer tokens
    - Including 10 or fewer words

Structural Similarity

```
public void startPrefixMapping(String prefix, String uri){
    List list = (List) prefixMapping.get(prefix);
    if (list == null) {
        list = new ArrayList();
        prefixMapping.put(prefix, list);
    }
    list.add(uri);
}
```

STEP1: a token sequence is generated from each method

List | list | = | (| List |) | prefixMapping.get | (| prefix |) | ;

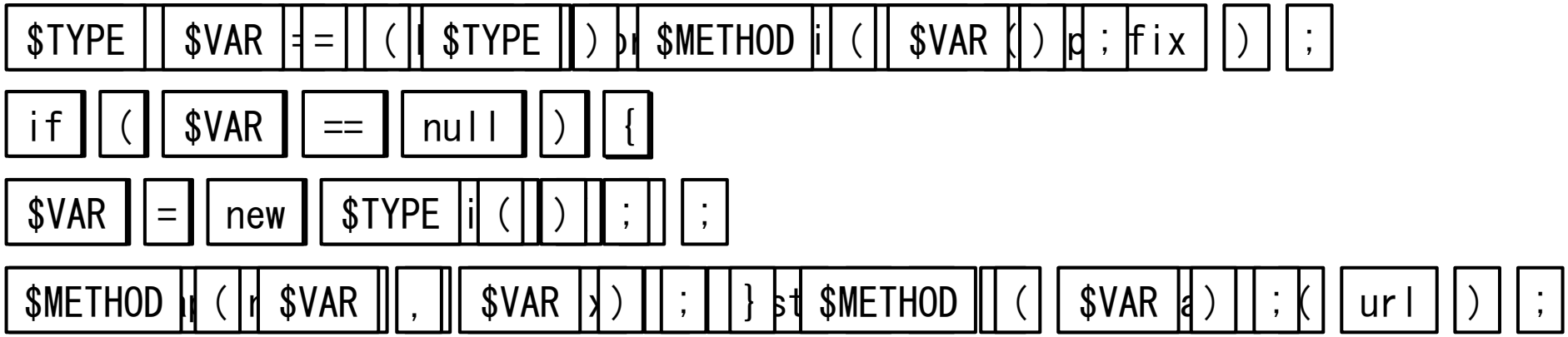
if | (| list | == | null |) | {

list | = | new | ArrayList | (|) | ;

prefixMapping.put | (| prefix | , | list |) | ; | } | list.add | (| url |) | ;

Structural Similarity

STEP2: all the tokens representing variable names, method names, and type names are replaced with special tokens. Three types of special tokens are all different from each other

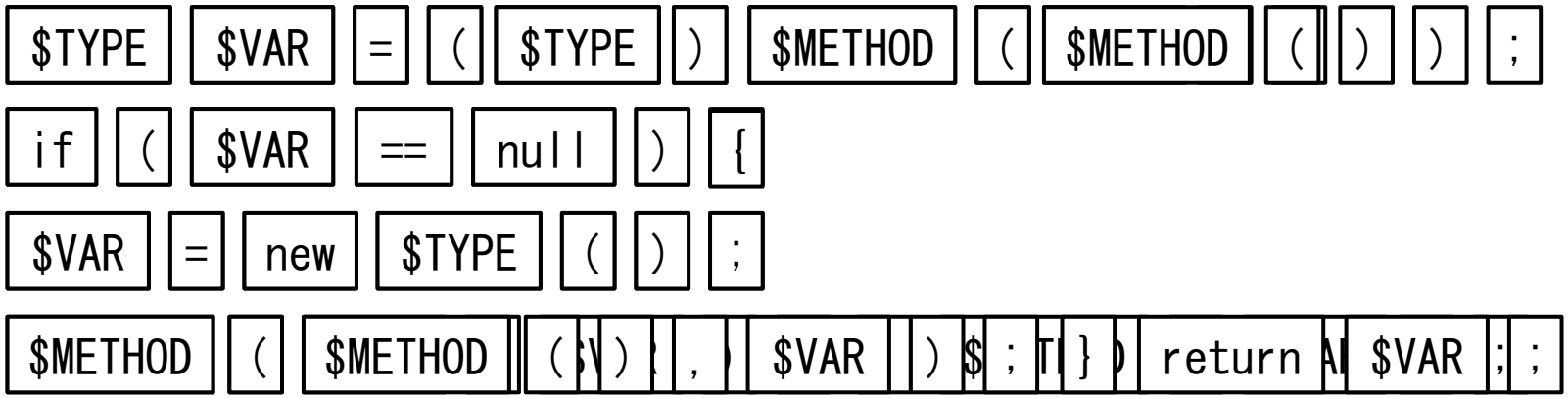


Structural Similarity

T_A

STEP3: the longest common subsequence between the two sequences is identified

T_B



Structural Similarity

STEP4: a quantified value of Structural Similarity (SS) is calculated

$$SS(T_A, T_B) = \min\left(\frac{|LCS(T_A, T_B)|}{|T_A|}, \frac{|LCS(T_A, T_B)|}{|T_B|}\right)$$

T_A

```
$TYPE $VAR = ( $TYPE ) $METHOD ( $VAR ) ;
if ( $VAR == null ) {
$VAR = new $TYPE ( ) ;
$METHOD ( $VAR , $VAR ) ; } $METHOD ( $VAR ) ;
```

T_B

```
$TYPE $VAR = ( $TYPE ) $METHOD ( $METHOD ( ) ) ;
if ( $VAR == null ) {
$VAR = new $TYPE ( ) ;
$METHOD ( $METHOD ( ) , $VAR ) ; } return $VAR ;
```

$$SS(T_A, T_B) = \min\left(\frac{33}{38}, \frac{33}{40}\right) = \frac{33}{40} = 0.825$$

Vocabulary Similarity

```
public void startPrefixMapping(String prefix, String uri){
    List list = (List) prefixMapping.get(prefix);
    if (list == null) {
        list = new ArrayList();
        prefixMapping.put(prefix, list);
    }
    list.add(uri);
}
```

STEP1: variable and method names are extracted

list | prefixMapping | get | prefix | put | add | url

STEP2: Nouns and verbs are obtained from the extracted names with their dictionary forms

list | map | get | prefix | put | add | url

Vocabulary Similarity

STEP3: a quantified value of Vocabulary Similarity (VS) is calculated

$$VS(V_A, V_B) = \frac{|V_A \cap V_B|}{|V_A \cup V_B|}$$

V_A list Map get prefix put add url

V_B thread stack get current put

$$VS(V_A, V_B) = \frac{2}{10} = 0.2$$

Method Name Similarity

- Quantifying a similarity of a whole signature is difficult

```
public void startPrefixMapping(String prefix, String uri){
    ...
```

- We adopt the simplest way
  - if names of two methods are exactly the same, their signatures are regarded as similar
  - if not, they are regarded as not similar

Structural Similarity

```
public void startPrefixMapping(String prefix, String uri){
    List list = (List) prefixMapping.get(prefix);
    if (list == null) {
        list = new ArrayList();
        prefixMapping.put(prefix, list);
    }
    list.add(uri);
}
```

STEP1: a token sequence is generated from each method

List | list | = | (| List |) | prefixMapping.get | (| prefix |) | ;

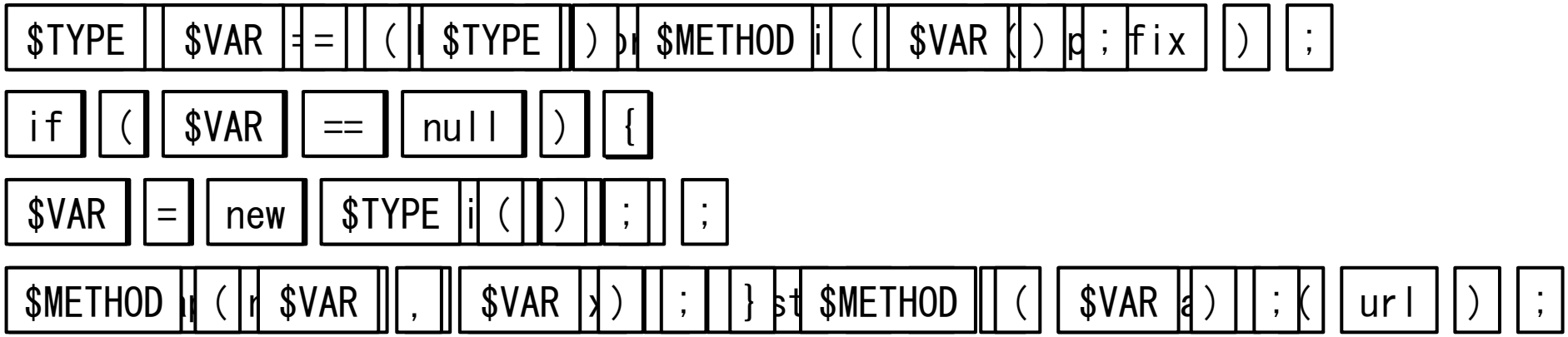
if | (| list | == | null |) | {

list | = | new | ArrayList | (|) | ;

prefixMapping.put | (| prefix | , | list |) | ; | } | list.add | (| url |) | ;

Structural Similarity

STEP2: all the tokens representing variable names, method names, and type names are replaced with special tokens. Three types of special tokens are all different from each other

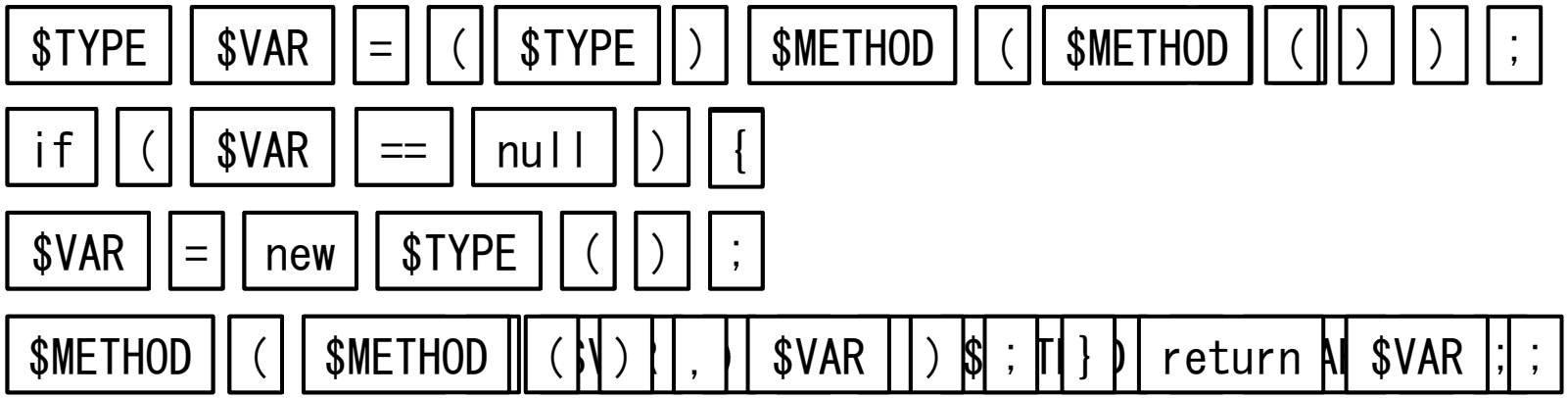


Structural Similarity

T_A

STEP3: the longest common subsequence between the two sequences is identified

T_B



Structural Similarity

STEP4: a quantified value of Structural Similarity (SS) is calculated

$$SS(T_A, T_B) = \min\left(\frac{|LCS(T_A, T_B)|}{|T_A|}, \frac{|LCS(T_A, T_B)|}{|T_B|}\right)$$

T_A

```
$TYPE $VAR = ( $TYPE ) $METHOD ( $VAR ) ;
if ( $VAR == null ) {
$VAR = new $TYPE ( ) ;
$METHOD ( $VAR , $VAR ) ; } $METHOD ( $VAR ) ;
```

T_B

```
$TYPE $VAR = ( $TYPE ) $METHOD ( $METHOD ( ) ) ;
if ( $VAR == null ) {
$VAR = new $TYPE ( ) ;
$METHOD ( $METHOD ( ) , $VAR ) ; } return $VAR ;
```

$$SS(T_A, T_B) = \min\left(\frac{33}{38}, \frac{33}{40}\right) = \frac{33}{40} = 0.825$$

Vocabulary Similarity

```
public void startPrefixMapping(String prefix, String uri){
    List list = (List) prefixMapping.get(prefix);
    if (list == null) {
        list = new ArrayList();
        prefixMapping.put(prefix, list);
    }
    list.add(uri);
}
```

STEP1: variable and method names are extracted

list | prefixMapping | get | prefix | put | add | url

STEP2: Nouns and verbs are obtained from the extracted names with their dictionary forms

list | map | get | prefix | put | add | url

Vocabulary Similarity

STEP3: a quantified value of Vocabulary Similarity (VS) is calculated

$$VS(V_A, V_B) = \frac{|V_A \cap V_B|}{|V_A \cup V_B|}$$

V_A list Map get prefix put add url

V_B thread stack get current put

$$VS(V_A, V_B) = \frac{2}{10} = 0.2$$

Method Name Similarity

- Quantifying a similarity of a whole signature is difficult

```
public void startPrefixMapping(String prefix, String uri){
    ...
```

- We adopt the simplest way
  - if names of two methods are exactly the same, their signatures are regarded as similar
  - if not, they are regarded as not similar