2IS55 Software Evolution

Code duplication and Program Differencing

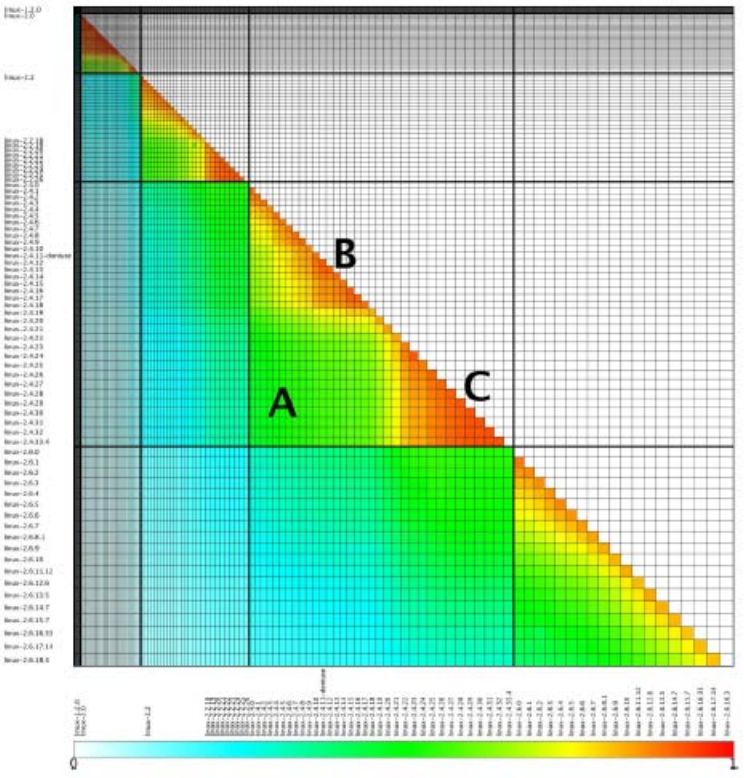
Alexander Serebrenik

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

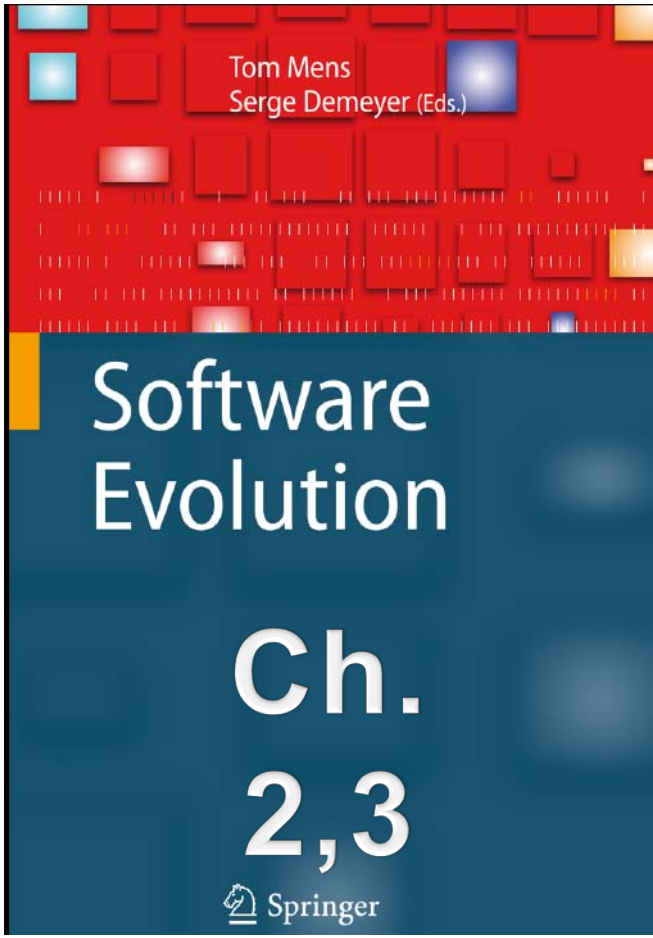
Assignment 4: Reminder

- **Deadline: March 30**
  - **Code duplication**
  - **Replication study of a scientific paper**
    - **Measure % clone coverage between subsequent versions of a system**
    - **Visualize using the heat map**
    - **Discuss the patterns observed**
  - **How is it going?**
    - **Postpone the deadline till April 6?**
    - **Mid-term week?**



May 4?

Sources



“Clone detection”

Rainer Koschke

http://www.informatik.uni-bremen.de/st/lehre/re09/softwareklone.pdf

“Program differencing”

Miryung Kim

Where are we now?

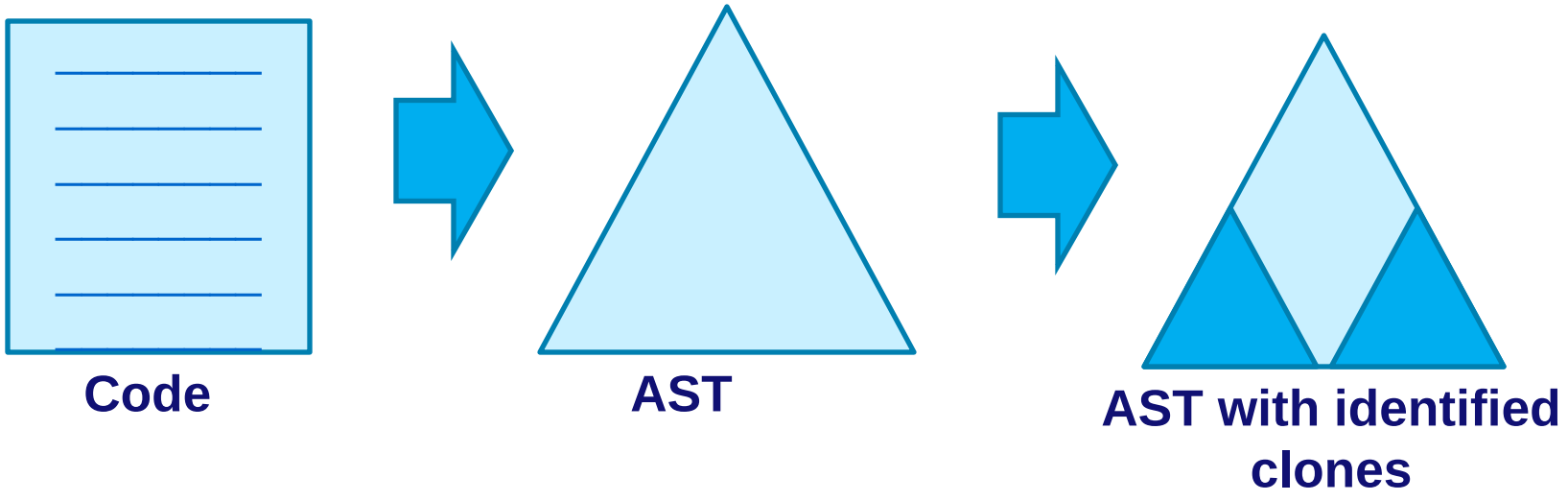
- Last week:
- Code cloning, code duplication, redundancy…
  - Type 1, 2, 3, 4 clones (more refined classif. possible)
  - Useful: reliability, reduced time, code preservation
  - Harmful: more interrelated code, more bugs
  - Ignore, eliminate, prevent, manage
  - Detection mechanisms
    - Text-based
    - Metrics-based
    - Token-based

Today

- Clone detection techniques
  - AST-based
    - [Baxter 1996]
    - AST+Tokens combined [Koschke et al. 2006]
  - Program Dependence Graph
    - [Krinke 2001]
  - Comparison of different techniques

AST-based clone detection [Baxter 1996]

- If we have a tokenizer we might also have a parser!
  - Applicability: the program should be parseable



- Compare every subtree with every other subtree?
  - For an AST of n nodes: $O(n^3)$
- Similarly to text: Partitioning with a hash function
  - Works for Type 1 clones

AST-based detection

- **Type 2**
  - **Either take a bad hash function ignoring small subtrees, e.g., names**
  - **Or replace identity by similarity**

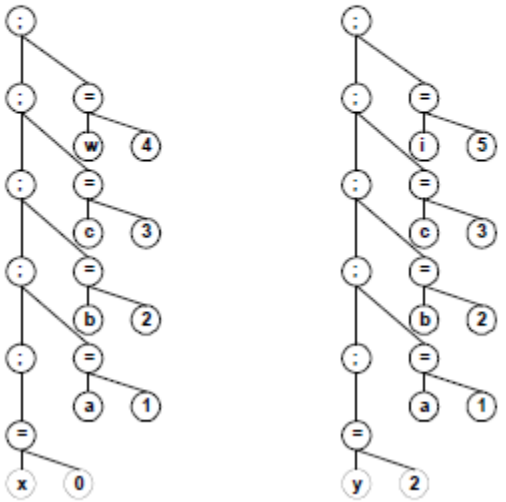
$$Similarity(T_1, T_2) = \frac{2 * Same(T_1, T_2)}{2 * Same(T_1, T_2) + Difference(T_1, T_2)}$$

- **Type 3**
  - **Sequences of subtrees**
  - **Go from Type 2-cloned subtrees to their parents**

- **Rather precise but still slow**

```
void f ()
{
  x=0;
  a=1;
  b=2;
  c=3;
  w=4;
}
```

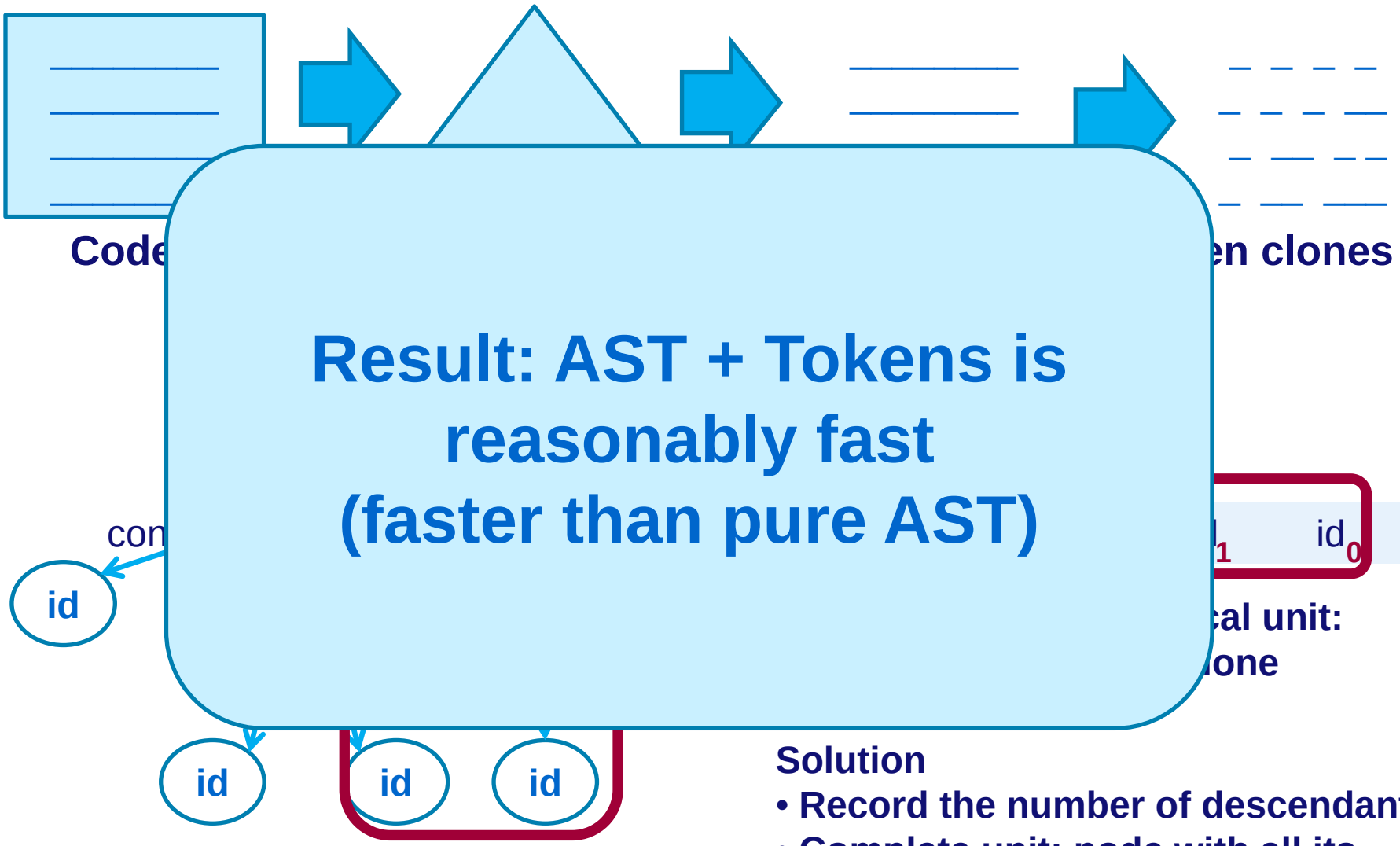
```
void g ()
{
  y=2;
  a=1;
  b=2;
  c=3;
  i=5;
}
```



Recapitulation from the last week

- [Baker 1995]
  - Token-based
  - Very fast:
    - 1.1 MLOC, minimal clone size: 30 LOC
    - 7 minutes on SGI IRIX 4.1, 40MHz, 256 MB
- [Baxter 1996]
  - AST-based
  - Precise but slow
- Idea: Combine the two! [Koschke et al. 2006]
  - In fact they do not use [Baker 1995] but a different token-based approach

AST + Tokens [Koschke et al. 2006]



Solution

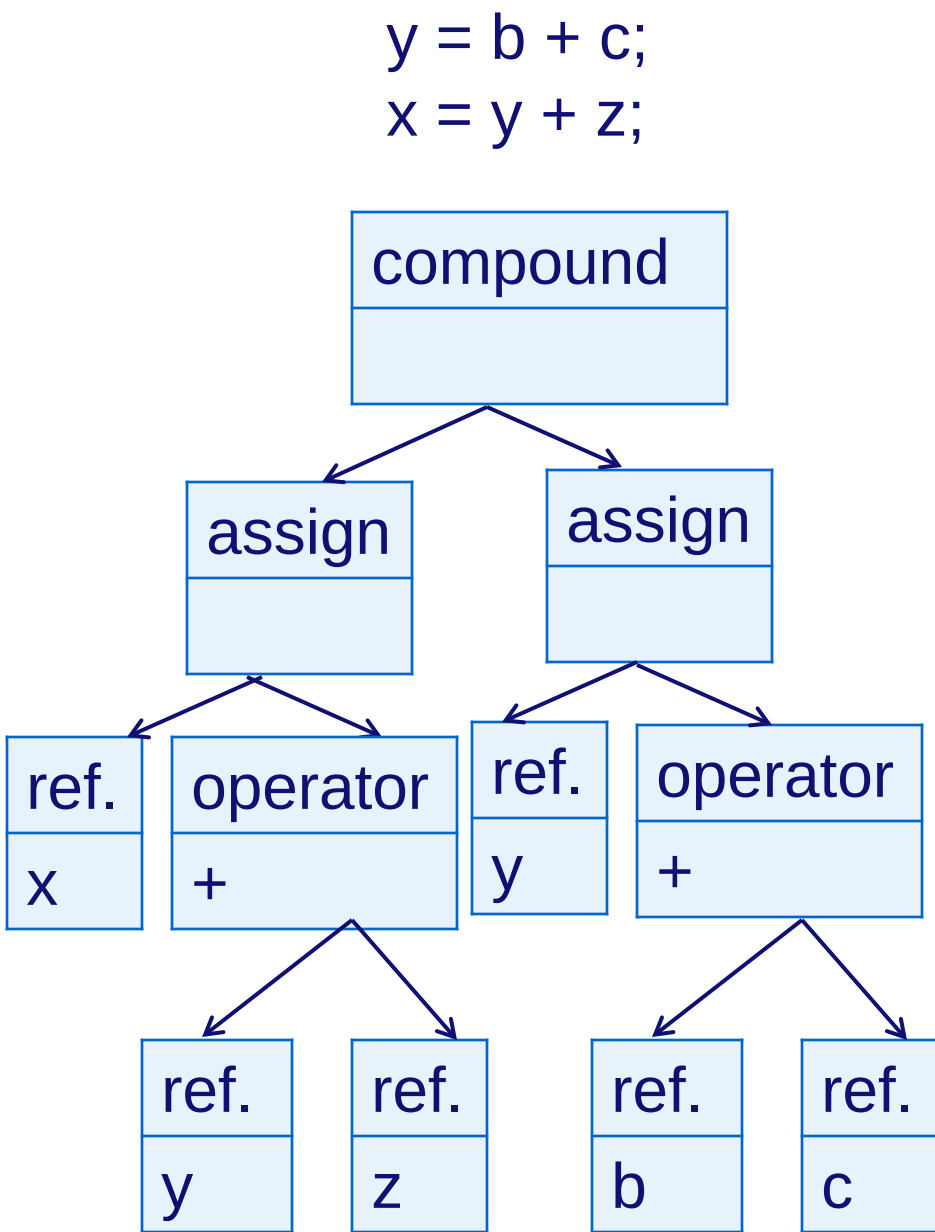
- **Record the number of descendants**
- **Complete unit: node with all its descendants**

Next step

- AST is a tree is a graph
- There are also other graph representations
  - Object Flow Graph (weeks 3 and 4)
  - UML class/package/… diagrams
  - Program Dependence Graph
- These representations do not depend on textual order
  - { x = 5; y = 7; } vs. { y = 7; x = 5; }

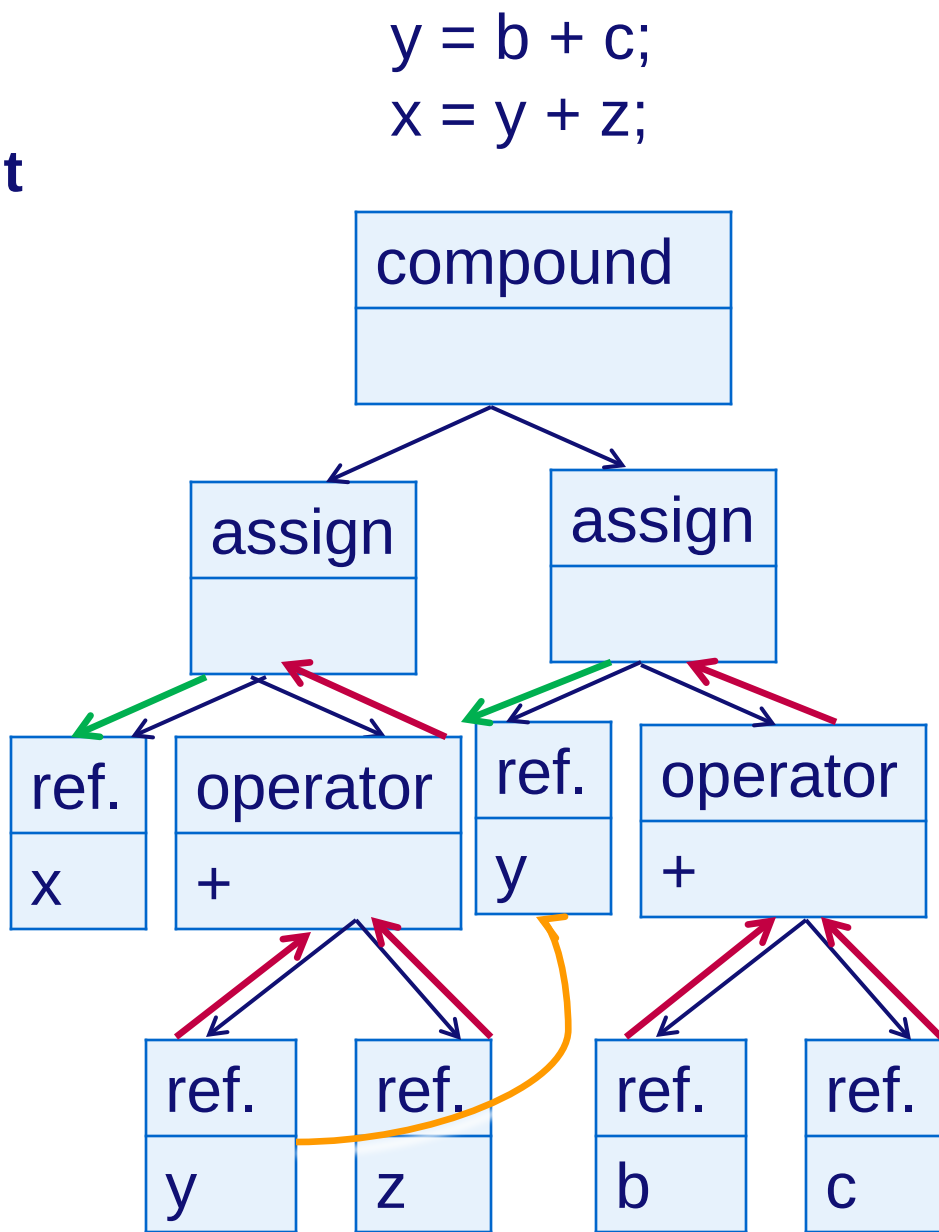
[Krinke 2001] PDG based

- **Vertices:**
  - **entry points, in- and output parameters**
  - **assignments, control statements, function calls**
  - **variables, operators**
- **Edges:**
  - **immediate dependencies**
    - **target has to be evaluated before the source**



[Krinke 2001] PDG based

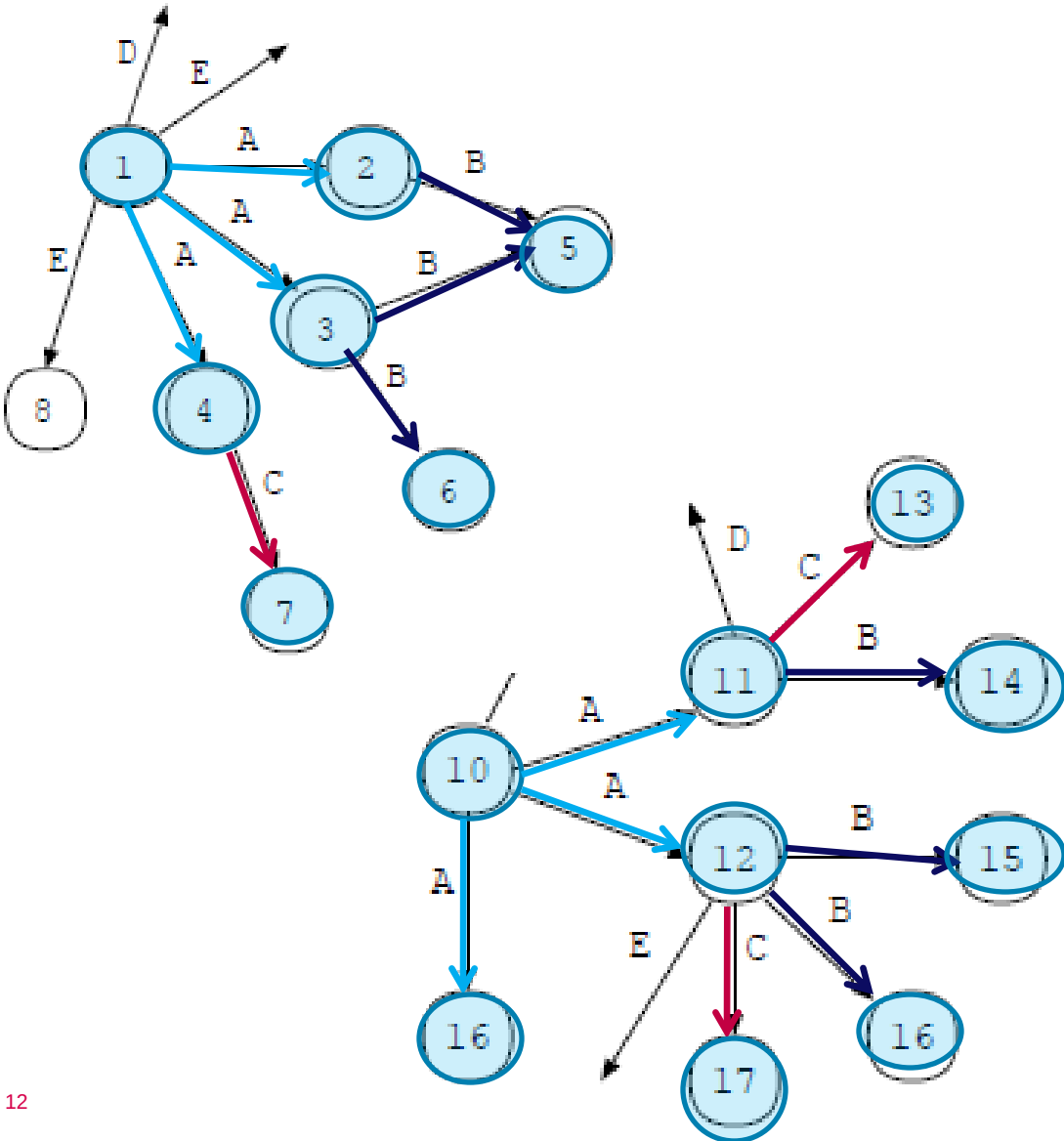
- **Vertices:**
  - **entry points, in- and output parameters**
  - **assignments, control statements, function calls**
  - **variables, operators**
- **Edges:**
  - **immediate dependencies**
  - **value dependencies**
  - **reference dependencies**
  - **data dependencies**
  - **control dependencies**
    - **Not in this example**



Identification of similar subgraphs – Theory

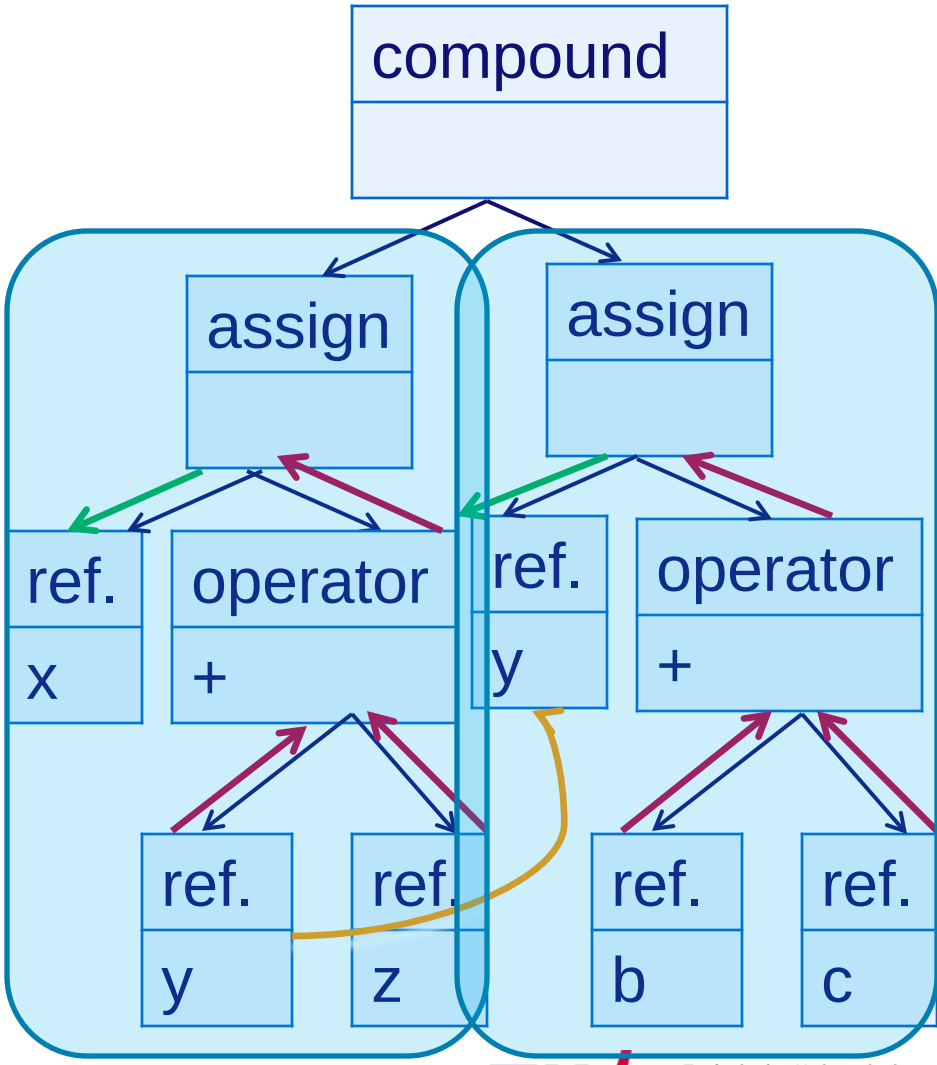
- Start with 1 and 10
- Partition the incident edges based on their labels
  - Select classes present in both graphs
- Add the target vertices to the set of reached vertices
- Repeat the process

- “Maximal similar subgraphs”



Identification of similar subgraphs – Practice

- **Sorts of edges are labels**
- **We also need to compare labels of vertices**
- **We should stop after k iterations**
  - **Higher k ⇒ higher recall**
  - **Higher k ⇒ higher execution time**
  - **Experiment: k = 20**



Choosing your tools: Precision / Recall

- **Quality depends on scenario [Type 1, Type 2, Type 3]**
- **[Roy *et al.* 2009]: 6 is maximal grade, 0 – minimal**

Tool	Technique	Category	S1	S2	S3
Duploc	Ducasse	Text	4	0	2.8
Marcus and Maletic			2.6	1.8	1.6
Dup	Baker	Token	4	2.8	0
CCFinder	Kamiya		5	3.8	0.8
CloneDr	Baxter	AST	6	4.3	3.8
cpdetector	Koschke		6	3.8	0
Mayrand		Metrics	3.3	4.8	3.4
Duplix	Krinke	Graph	5	4.8	4

More tools: ConQAT, DECKARD, Dude, Simian

Which technique/tool is the best one?

- Quality
  - Precision
  - Recall
- Usage
  - Availability
  - Dependence on a platform
  - Dependence on an external component (lexer, tokenizer, …)
  - Input/output format
- Programming language
- Clones
  - Granularity
  - Types
  - Pairs vs. groups
- Technique
  - Normalization
  - Storage
  - Worst-case complexity
  - Pre-/postprocessing
- Validation
- *Extra: metrics*

Clone detection techniques: Summary

- Many different techniques
  - Text, metrics, tokens, AST, program dependence graph, combinations
- Techniques are often supported by tools
- Precision depends on what kind of clones we need:
  - Type 1, Type 2, Type 3, Type 4
- Extra conditions
  - Programming language, presence of external tools, platforms, extra's (metrics), normalization, ...

Program differencing: Remote cousin of cloning

- Why?
  - Version control
    - What have we changed since…
    - How can we merge the changes by Alice and Bob?
  - What do we need to retest?
  - What is the impact of our change?
- Formally
  - Input: Two programs
  - Output:
    - Differences between the two programs
    - Unchanged code fragments in the old version and their corresponding locations in the new
- Similar to clone detection
  - Comparison of lines, tokens, trees and graphs

Diff: Longest common subsequence

- Program: sequence of lines
- Object of comparison: line

- Comparison:
  - 1:1
  - lines are identical
  - matched pairs cannot overlap

- Technique: longest common subsequence
  - Minimal number of additions/deletions steps
  - Dynamic programming

Longest common subsequence

- **Programs X (n lines), Y (m lines)**
- **Data structure C[0..n,0..m]**
- **Init: C[r,0]=0, C[0,c]=0 for any r and c**

```
p0 mA (){
p1   if (pred_a) {
p2       foo()
p3   }
p4 }
```

X

```
c0 mA (){
c1   if (pred_a0) {
c2       if (pred_a) {
c3           foo()
c4       }
c5   }
c6 }
```

Y

C		c0	c1	c2	c3	c4	c5	c6
	0	0	0	0	0	0	0	0
p0	0							
p1	0							
p2	0							
p3	0							
p4	0							

Longest common subsequence

- **For every r and every c**
  - **If X[r]=Y[c] then C[r,c]=C[r-1,c-1]+1**
  - **Else C[r,c]=max(C[r,c-1],C[r-1,c])**

```
p0 mA (){
p1   if (pred_a) {
p2       foo()
p3   }
p4 }
```

X

```
c0 mA (){
c1   if (pred_a0) {
c2       if (pred_a) {
c3           foo()
c4       }
c5   }
c6 }
```

Y

C		c 0	c 1	c 2	c 3	c 4	c 5	c 6
	0	0	0	0	0	0	0	0
p0	0	**1**	1	1	1	1	1	1
p1	0	1	1	**2**	2	2	2	2
p2	0	1	1	2	**3**	3	3	3
p3	0	1	1	2	3	**4**	**4**	**4**
p4	0	1	1	2	3	**4**	**5**	**5**

Longest common subsequence

- **Start with r=n and c=m**
- **backTrace(r,c)**
  - **If r=0 or c=0 then “”**
  - **If X[r]=Y[c] then backTrace(r-1,c-1)+X[r]**
  - **Else**
    - **If C[r,c-1] > C[r-1,c] then backTrace(r,c-1) else backTrace(r-1,c)**

```
p0 mA (){
p1   if (pred_a) {
p2       foo()
p3   }
p4 }

X
```

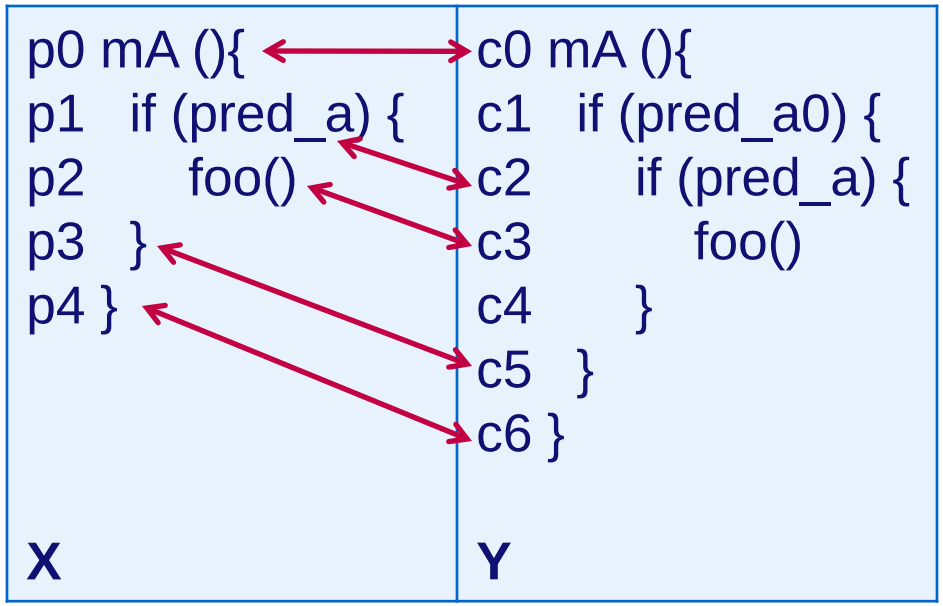
```
c0 mA (){
c1   if (pred_a0) {
c2       if (pred_a) {
c3           foo()
c4       }
c5   }
c6 }

Y
```

C		c0	c1	c2	c3	c4	c5	c6
	0	0	0	0	0	0	0	0
p0	0	**1**	1	1	1	1	1	1
p1	0	1	1	**2**	2	2	2	2
p2	0	1	1	2	**3**	3	3	3
p3	0	1	1	2	3	**4**	**4**	**4**
p4	0	1	1	2	3	**4**	**5**	**5**

Longest common subsequence

- **Start with r=n and c=m**
- **backTrace(r,c)**
  - **If r=0 or c=0 then “”**
  - **If X[r]=Y[c] then backTrace(r-1,c-1)+X[r]**
  - **Else**
    - **If C[r,c-1] > C[r-1,c] then backTrace(r,c-1) else backTrace(r-1,c)**



C		c 0	c 1	c 2	c 3	c 4	c 5	c 6
	0	0	0	0	0	0	0	0
p0	0	**1**	1	1	1	1	1	1
p1	0	1	1	**2**	2	2	2	2
p2	0	1	1	2	**3**	3	3	3
p3	0	1	1	2	3	**4**	**4**	**4**
p4	0	1	1	2	3	**4**	**5**	**5**

Diff: Summarizing

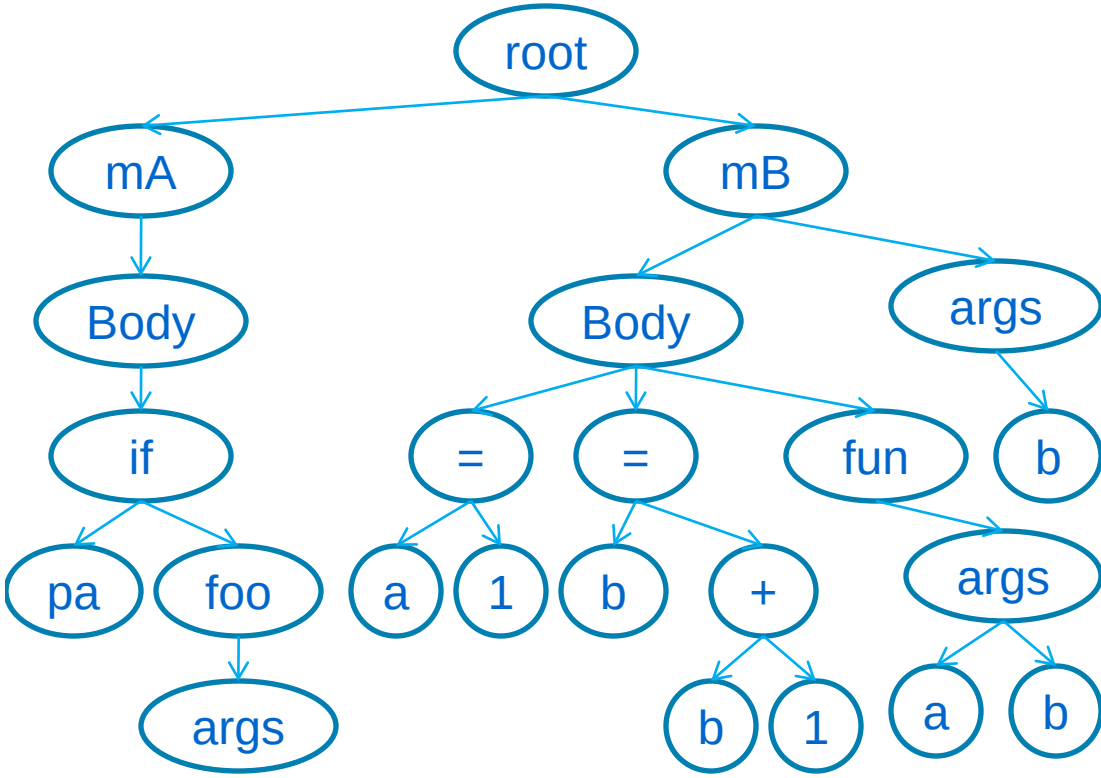
- **Comparison:**
  - **1:1, identical lines, non-overlapping pairs**
- **Technique: longest common subsequence**
- **What kind of code modifications will diff miss?**
  - **Copy & paste: apple ⇒ applple**
    - **1:1 is violated**
  - **Move: apple ⇒ aplep**

More than lines: AST Diff [Yang 1992]

- **Construct ASTs for the input programs**

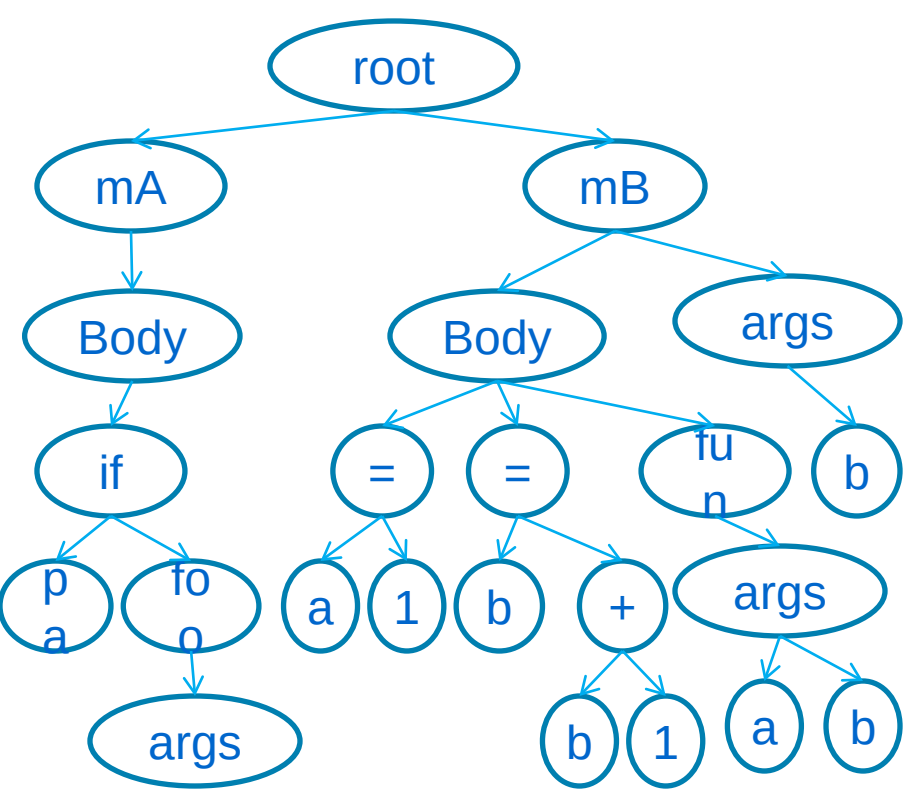
```
p0 mA (){
p1   if (pa) {
p2       foo()
p3   }
p4 }
p5 mB (b) {
p6   a = 1
p7   b = b+1
p8   fun(a,b)
p9 }
```

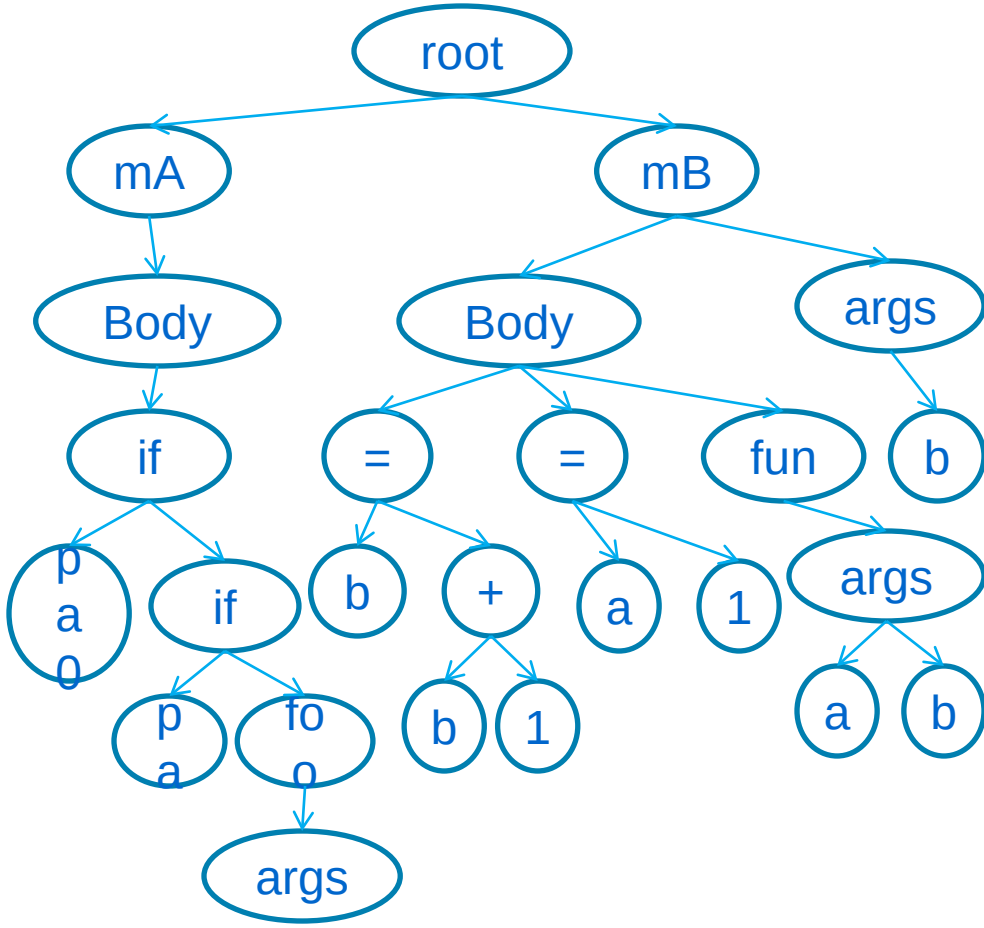
x



More than lines: AST Diff [Yang 1992]

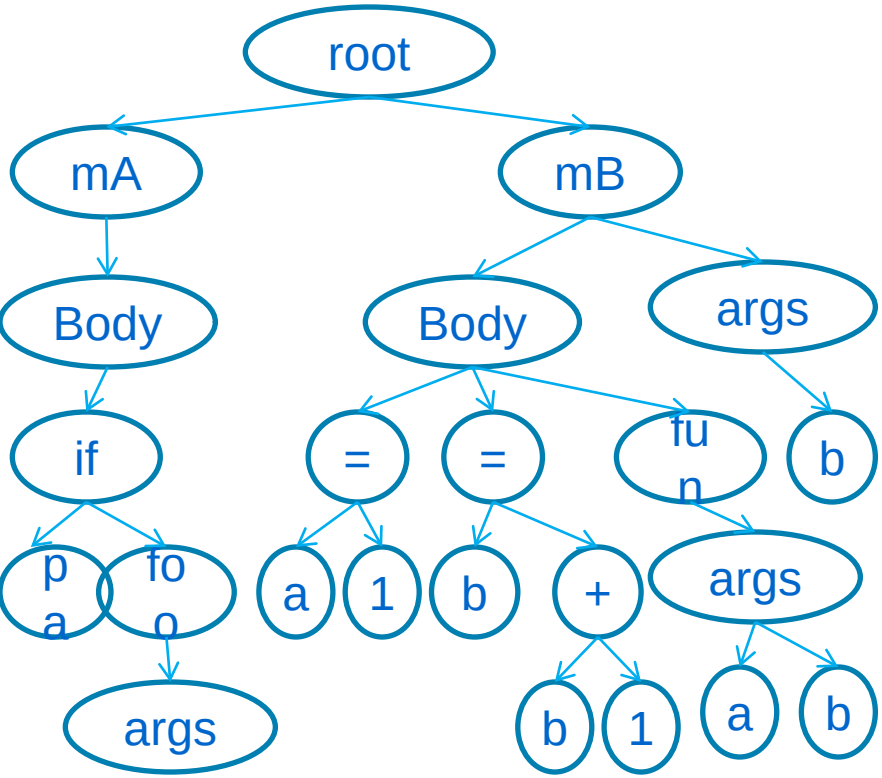
- **Recursive algo pairwise subtree comparison**

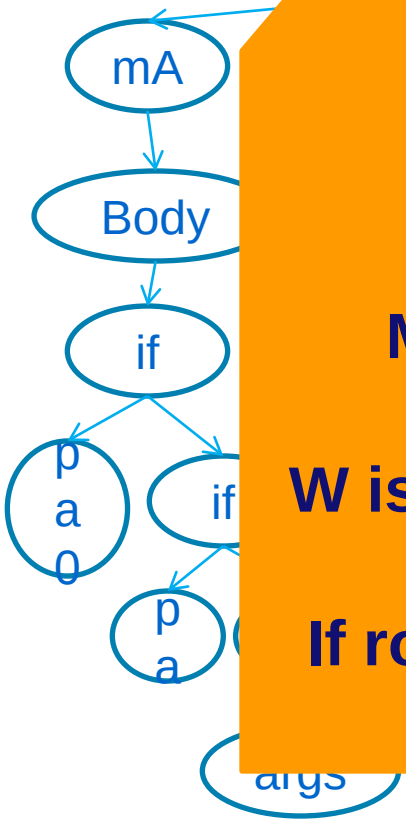




- **n – first level subtrees in X**
- **m – first level subtrees in Y**
- **Array: M[0..n, 0..m]**

More than lines: AST Diff [Yang 1992]





M[i,j] = max(M[i,j-1], M[i-1,j], M[i-1,j-1]+W[i,j])

W is the recursive call

If root symbols differ return 0

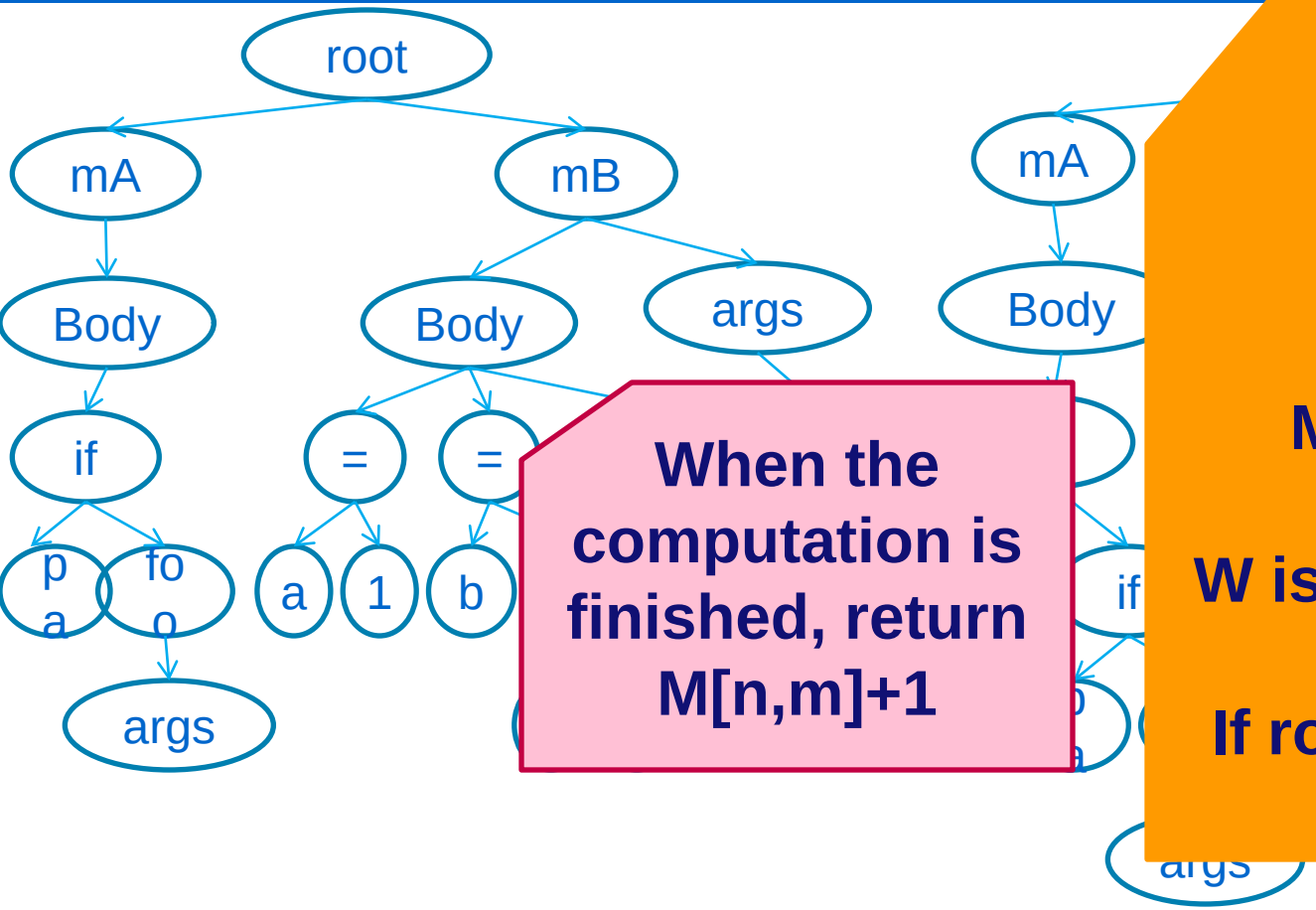
M	0	mA	mB
0	0	0	0
mA	0		
mB	0		

M	0	Body
0	0	0
Body	0	

M	0	if
0	0	0
if	0	

M	0	pa0	if
0	0	0	0
pa	0		
foo	0		

More than lines: AST Diff [Yang 1992]



When the computation is finished, return M[n,m]+1

M[i,j] = max(M[i,j-1], M[i-1,j], M[i-1,j-1]+W[i,j])

W is the recursive call

If root symbols differ return 0

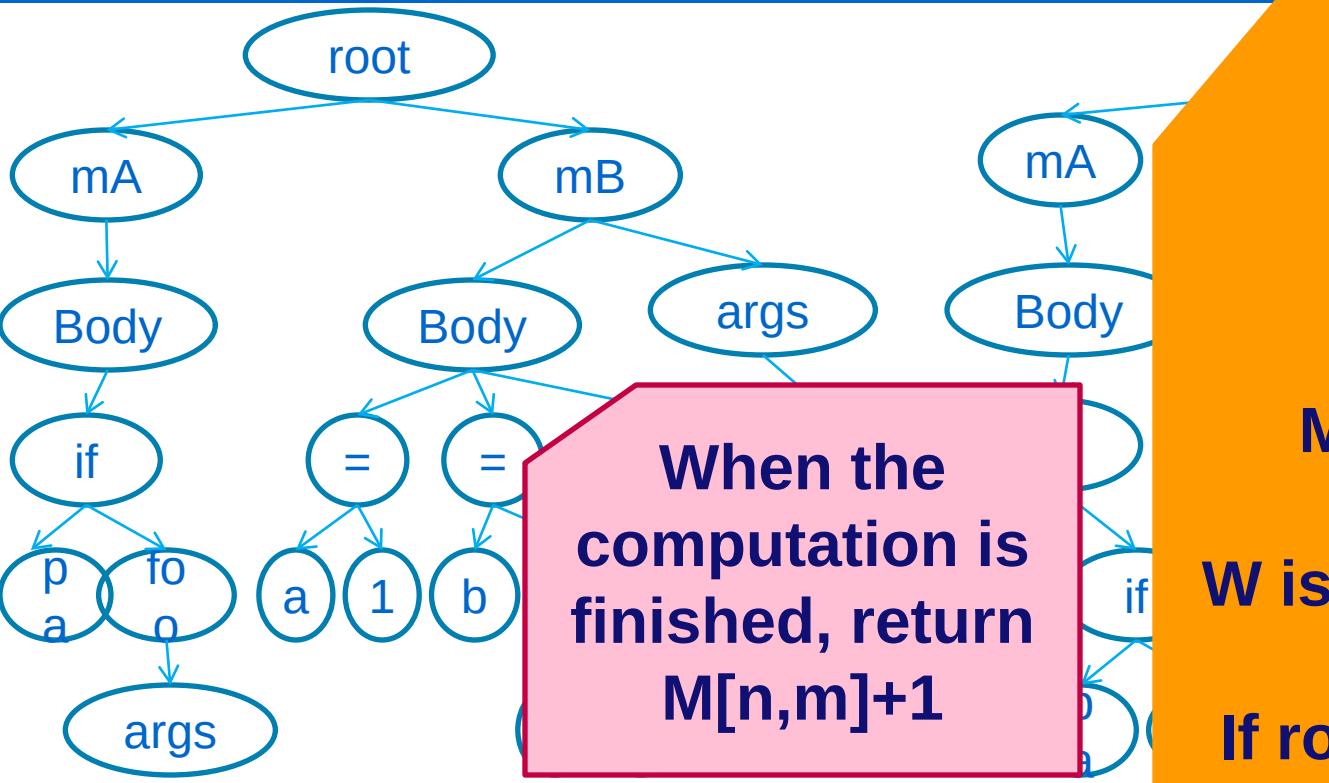
M	0	mA	mB
0	0	0	0
mA	0		
mB	0		

M	0	Body
0	0	0
Body	0	

M	0	if
0	0	0
if	0	1

M	0	pa0	if
0	0	0	0
pa	0	0	0
foo	0	0	0

More than lines: AST Diff [Yang 1992]



When the computation is finished, return M[n,m]+1

M[i,j] = max(M[i,j-1], M[i-1,j], M[i-1,j-1]+W[i,j])

W is the recursive call

If root symbols differ return 0

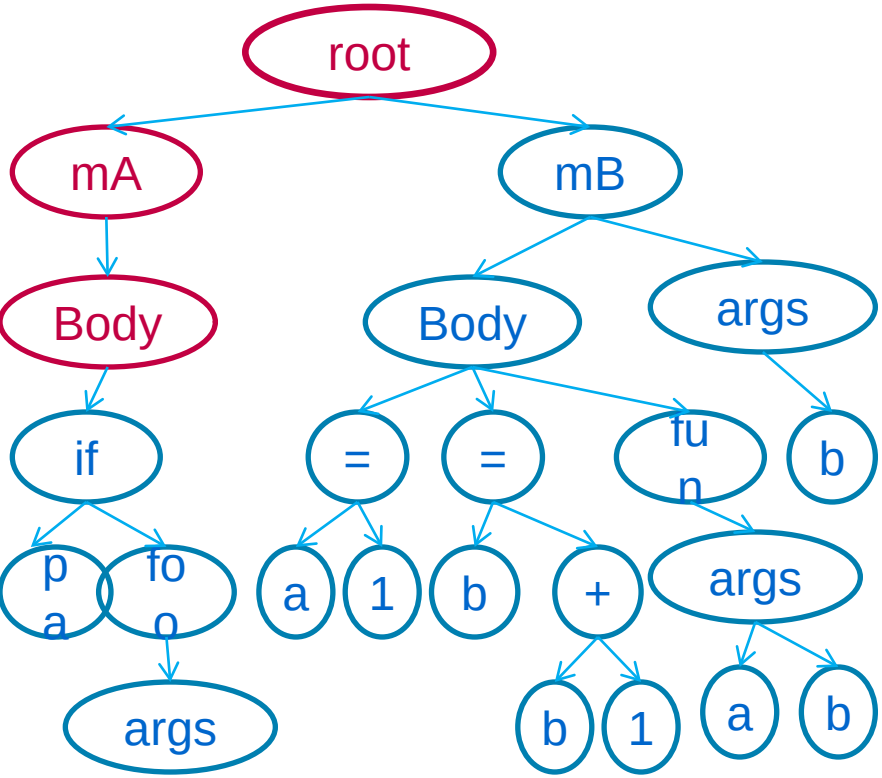
M	0	mA	mB
0	0	0	0
mA	0	**3**	
mB	0		

M	0	Body
0	0	0
Body	0	**2**

M	0	if
0	0	0
if	0	**1**

M	0	pa0	if
0	0	0	0
pa	0	0	0
foo	0	0	0

More than lines: AST Diff [Yang 1992]



p0 mA (){	c0 mA (){
p1 if (pa) {	c1 if (pa0) {
p2 foo()	c2 if (pa) {
p3 }	c3 foo()
p4 }	c4 }
p5 mB (b) {	c5 }
p6 a = 1	c6 }
p7 b = b+1	c7 mB (b) {
p8 fun(a,b)	c8 b = b+1
p9 }	c9 a = 1
	c10 fun(a,b)
	c11 }
X	**Y**

M	0	mA	mB
0	0	0	0
mA	0	**3**	
mB	0		

Continuing the process

```
p0 mA (){
p1   if (pa) {
p2       foo()
p3   }
p4 }
p5 mB (b) {
p6   a = 1
p7   b = b+1
p8   fun(a,b)
p9 }
```

X

```
c0 mA (){
c1   if (pa0) {
c2       if (pa) {
c3           foo()
c4       }
c5   }
c6 }
c7 mB (b) {
c8     b = b+1
c9     a = 1
c10    fun(a,b)
c11 }
```

Y

- **Advantages**
  - **Respect the parent-child relations**
  - **Ignore the order between siblings**
- **Disadvantages**
  - **Sensitive to tree level changes (“if (pa)”)**
  - **Ignore dependencies such as data flow, etc**

- **Can be adapted to OO-specific dataflow (inheritance, exceptions): JDiff**

Changes never come alone

- **Search and replace**
- **Check-in comment: “Common methods go in an abstract class. Easier to extend/maintain/fix”**

- **Change: a rule rather than a set application results**
  - **Rules can have exceptions**

- **Idea [Kim and Notkin 2009]**
  - **Observe differences between subsequent versions**
  - **Formalize them as facts**
  - **Discover rules (à la data mining)**
  - **Record exceptions**

Structural changes [Kim and Notkin 2009]

- **Program ⇒ collection of facts**

v_1

```
class Bus {
    void start(Key c) {
        c.on = false; }
}

class Key {
    boolean on = false;
    void chk (Key c) {…}
    void out() {…}
}
```

FB_0

```
type(“Bus”)
method(“Bus.start”,”start”,”Bus”)
access(“Key.on”,”Bus.start”)
type(“Key”)
field(“Key.on”, “on”, “Key”)
method(“Key.chk”,”chk”,”Key”)
method(“Key.out”,”out”,”Key”)
```

v_2

```
class Bus {
    void start(Key c) {
        log(); }
}

class Key {
    boolean on = false;
    void chk (Key c) {…}
    void output() {…}
}
```

FB_n

```
type(“Bus”)
method(“Bus.start”,”start”,”Bus”)
calls(”Bus.start”,”log”)
type(“Key”)
field(“Key.on”, “on”, “Key”)
method(“Key.chk”,”chk”,”Key”)
method(“Key.output”,”output”,”Key”)
```

Structural changes [Kim and Notkin 2009]

- **Calculate the differences**

```
type(“Bus”)
method(“Bus.start”,”start”,”Bus”)
access(“Key.on”,”Bus.start”)
type(“Key”)
field(“Key.on”, “on”, “Key”)
method(“Key.chk”,”chk”,”Key”)
method(“Key.out”,”out”,”Key”)
```

```
type(“Bus”)
method(“Bus.start”,”start”,”Bus”)
calls(”Bus.start”,”log”)
type(“Key”)
field(“Key.on”, “on”, “Key”)
method(“Key.chk”,”chk”,”Key”)
method(“Key.output”,”output”,”Key”)
```

ΔFB

```
deleted_access(“Key.on”,“Bus.start”)
added_calls(”Bus.start”,”log”)
deleted_method(“Key.out”,”out”,”Key”)
added_method(“Key.output”,”output”,”Key”)
```

Deleting Key.out and adding Key.output can be identified as renaming and removed.

Learn the rules

- **Rule inference should depend on ΔFB**
  - **Is this enough? Why?**
  - **Context information is lost!**
    - **We need FB_0 and FB_n**
    - **To distinguish between the facts: past_ and current_**

```
past_type(“Bus”)
past_method(“Bus.start”,”start”,”Bus”)
past_access(“Key.on”,”Bus.start”)
past_type(“Key”)
past_field(“Key.on”, “on”, “Key”)
past_method(“Key.chk”,”chk”,”Key”)
past_method(“Key.out”,”out”,”Key”)

current_type(“Bus”)
current_method(“Bus.start”,”start”,”Bus”)
current_calls(”Bus.start”,”log”)
current_type(“Key”)
current_field(“Key.on”, “on”, “Key”)
current_method(“Key.chk”,”chk”,”Key”)
current_method(“Key.output”,”output”,”Key”)

deleted_access(“Key.on”,“Bus.start”)
added_calls(”Bus.start”,”log”)
```

Rules

- **Datalog style**
- **Restrictions:**
  - **Antecedent: one type of facts**
  - **Consequent: only deleted_ or added_**

past_*	⇒	deleted_*	feature or dependency removal
past_*	⇒	added_*	consistent clone update: all code elements with similar characteristics in the old version added similar code
current_*	⇒	added_*	feature addition
deleted_*	⇒	added_*	API migration
added_*	⇒	deleted_*	

How are the rules learned?

Algorithm 1: LSdiff Rule Inference Algorithm

Input: FB_o, FB_n, ΔFB, m, a, k, and β
Output: L and U
/* Initialize R, a set of ungrounded rules; L, a set of learned rules; and U, a set of facts in ΔFB that are not covered by L. */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules (ΔFB, FB_o, FB_n); /* reduce ΔFB with default winnowing rules. */
R := createInitialRules (m); /* create rules with an empty antecedent by enumerating all possible consequents. */

- **Winnowing rules remove trivial dependencies**
  - **“if a class is deleted, so are all its methods”**
- **Ungrounded – with variables**

U

deleted_access(“Key.on”,“Bus.start”)
added_calls(”Bus.start”,”log”)

R *createInitialRules*

deleted_access(f,m)
added_calls(m1,m2)

How are the rules learned?

Algorithm 1: LSdiff Rule Inference Algorithm

Input: FB_o, FB_n, ΔFB, m, a, k, and β
Output: L and U
/* Initialize R, a set of ungrounded rules; L, a set of learned rules; and U, a set of facts in ΔFB that are not covered by L. */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules (ΔFB, FB_o, FB_n); /* reduce ΔFB with default winnowing rules. */
R := createInitialRules (m); /* create rules with an empty antecedent by enumerating all possible consequents. */
foreach $i = 1 \ldots k$ **do**
  R := extendUngroundedRules (R) ; /* extend all ungrounded rules in R by adding all possible literals to their antecedent. */

- **Winnowing rules remove trivial dependencies**
  - **“if a class is deleted, so are all its methods”**
- **Ungrounded – with variables**

R *extendUngroundedRules*

deleted_access(f,m) ⇐
  past_method(m, mclass, mshort)
deleted_access(f,m) ⇐
  past_field(f, fshort, fclass)
deleted_access(f,m) ⇐
  past_access(f,m)

How are the rules learned?

Algorithm 1: LSdiff Rule Inference Algorithm

Input: FB_o, FB_n, ΔFB, m, a, k, and β
Output: L and U

```
/* Initialize R, a set of ungrounded rules; L,
   a set of learned rules; and U, a set of
   facts in ΔFB that are not covered by L.  */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules (ΔFB, FB_o,
FB_n);     /* reduce ΔFB with default winnowing
rules.  */
R := createInitialRules (m);     /* create rules
with an empty antecedent by enumerating all
possible consequents.  */
foreach i = 1 ... k do
    R := extendUngroundedRules (R) ;     /* extend
    all ungrounded rules in R by adding all
    possible literals to their antecedent.  */
    foreach r ∈ R do
        G := createPartiallyGroundedRules (r) ;
        /* try all possible constant
        substitutions for r's variable.  */
```

r deleted_access(f,m) ⇐
  past_access(f,m)

G

deleted_access (“Key.on”,”Bus.start”) ⇐
  past_access (“Key.on”,”Bus.start”)
deleted_access (“Key.on”,”Key.chk”) ⇐
  past_access (“Key.on”,”Key.chk”)
deleted_access (“Key.on”,”Key.out”) ⇐
  past_access (“Key.on”,”Key.out”)
deleted_access (“Key.on”,m) ⇐
  past_access (“Key.on”,m)
deleted_access (f,”Bus.start”) ⇐
  past_access (f,”Bus.start”)
deleted_access (f,”Key.chk”) ⇐
  past_access (f,”Key.chk”)
deleted_access (f,”Key.out”) ⇐
  past_access (f,”Key.out”)
deleted_access(f,m) ⇐
  past_access(f,m)

How are the rules learned?

Algorithm 1: LSdiff Rule Inference Algorithm

Input: FB_o, FB_n, ΔFB, m, a, k, and β
Output: L and U

```
/* Initialize R, a set of ungrounded rules; L,
   a set of learned rules; and U, a set of
   facts in ΔFB that are not covered by L.  */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules (ΔFB, FB_o,
FB_n);     /* reduce ΔFB with default winnowing
rules.  */
R := createInitialRules (m);     /* create rules
with an empty antecedent by enumerating all
possible consequents.  */
foreach i = 1 ... k do
    R := extendUngroundedRules (R) ;     /* extend
    all ungrounded rules in R by adding all
    possible literals to their antecedent.  */
    foreach r ∈ R do
        G := createPartiallyGroundedRules (r) ;
        /* try all possible constant
        substitutions for r's variable.  */
        foreach g in G do
            if isValid (g) then
                L :=L ∪ {g};
                U :=U − {g.matches};
            end
        end
    end
    R :=selectRules (R, β) ; /* select the best β
    rules in R  */
end
```

- **Validity**
  - **≥ m matching facts**
  - **Match/(Match+NonMatch) ≥ a**

R (examples)

deleted_access (“Key.on”,”Bus.start”)
⇐ past_access (“Key.on”,”Bus.start”)
One fact, 100% precision

deleted_access (“Key.on”,”Key.chk”) ⇐
past_access (“Key.on”,”Key.chk”)
No matching facts

deleted_access (“Key.on”,m) ⇐
past_access (“Key.on”,m)
One fact, 100% precision

deleted_access (f,”Bus.start”) ⇐
past_access (f,”Bus.start”)
One fact, 100% precision

deleted_access(f,m) ⇐ past_access(f,m)
One fact, 100% precision

How are the rules learned?

Algorithm 1: LSdiff Rule Inference Algorithm

Input: FB_o, FB_n, ΔFB, m, a, k, and β
Output: L and U
/* Initialize R, a set of ungrounded rules; L, a set of learned rules; and U, a set of facts in ΔFB that are not covered by L. */
R := ∅, L := ∅, U := ΔFB;
U := applyDefaultWinnowingRules (ΔFB, FB_o, FB_n); /* reduce ΔFB with default winnowing rules. */
R := createInitialRules (m); /* create rules with an empty antecedent by enumerating all possible consequents. */
foreach $i = 1 \ldots k$ **do**
  R := extendUngroundedRules (R) ; /* extend all ungrounded rules in R by adding all possible literals to their antecedent. */
  foreach $r \in$ R **do**
    G := createPartiallyGroundedRules (r) ;
    /* try all possible constant substitutions for r's variable. */
    foreach g *in* G **do**
      if isValid (g) **then**
        L :=L ∪ {g};
        U :=U − {g.matches};
      end
    end
  end
  R :=selectRules (R, β) ; /* select the best β rules in R */
end

- For more realistic examples
  - Thresholds do matter
  - More general = less accurate
- At the next step more antecedents are added, e.g.,
  - past_method(m,"start",t) ∧ past_subtype("Car",t) ⇒ added_calls(m,"Key.chk")
- β controls the number of rules kept for extension at the next iteration

Evaluation

- **On average, 75% of facts in ΔFB are covered by inferred rules**
  - **~75% of structural differences are systematic change.**
- **Concise:**
  - **Textual diff: 997 lines, 16 files**
  - **LSdiff: 7 rules and 27 facts**

How did the users experience the tool?

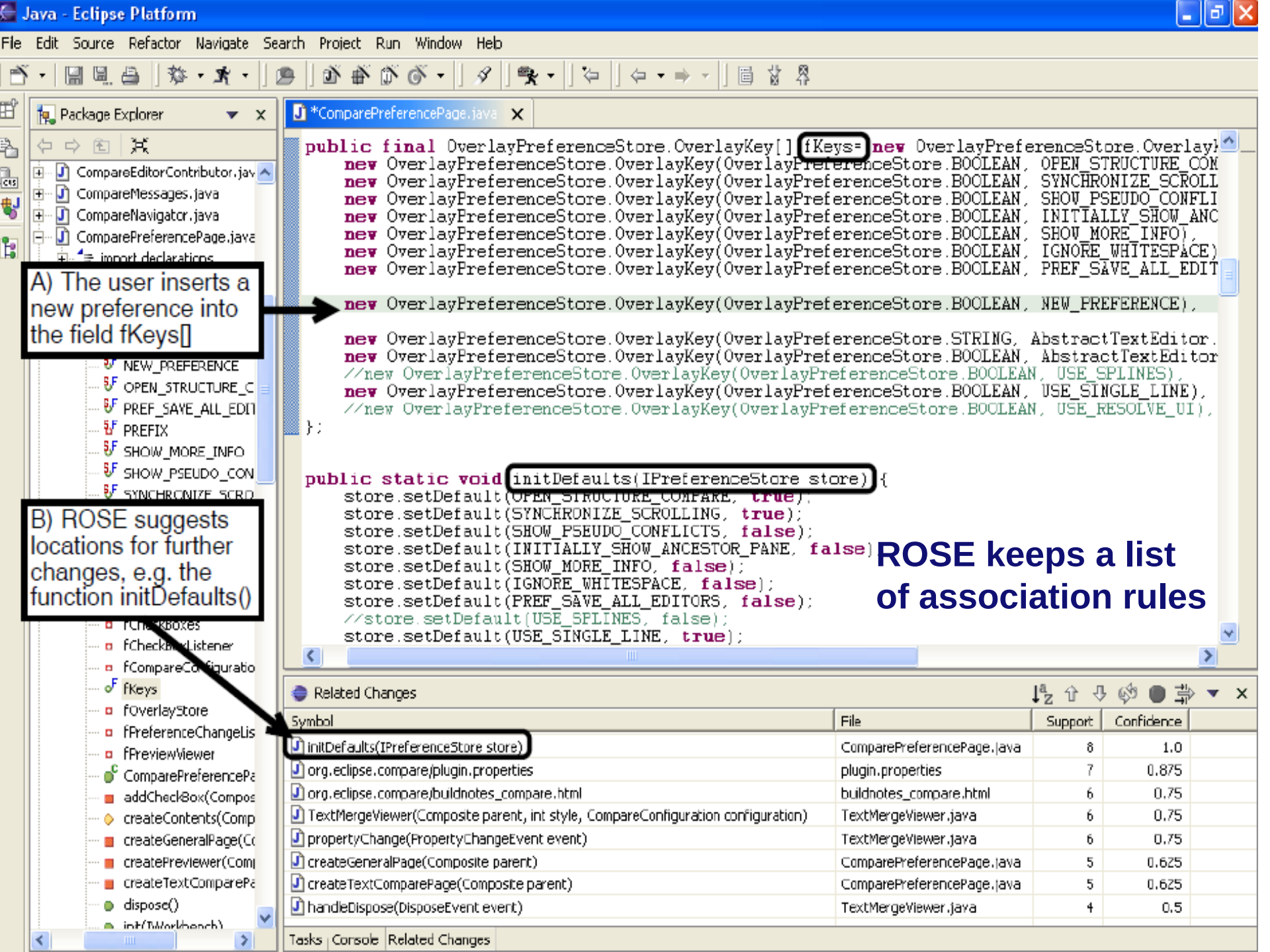
- **Positive**
  - **“You can’t infer the intent of a programmer, but this is pretty close.”**
  - **“This ‘except’ thing is great!”**

- **Negative**
  - **“This looks great for big architectural changes, but I wonder what it would give you if you had lots of random changes.”**
  - **“This will look for relationships that do not exist.”**

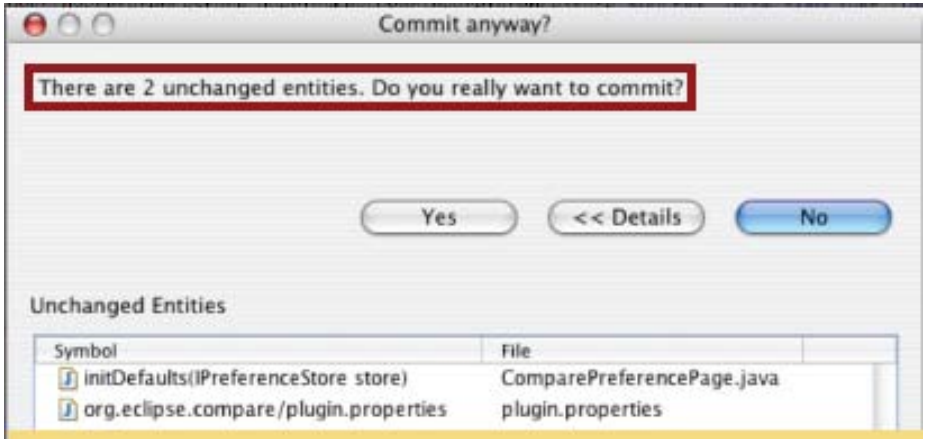
What about “random” changes?

- **eROSE [Zimmermann, Weißgerber, Diehl, Zeller ‘04]**

Developers who modified this function also modified…

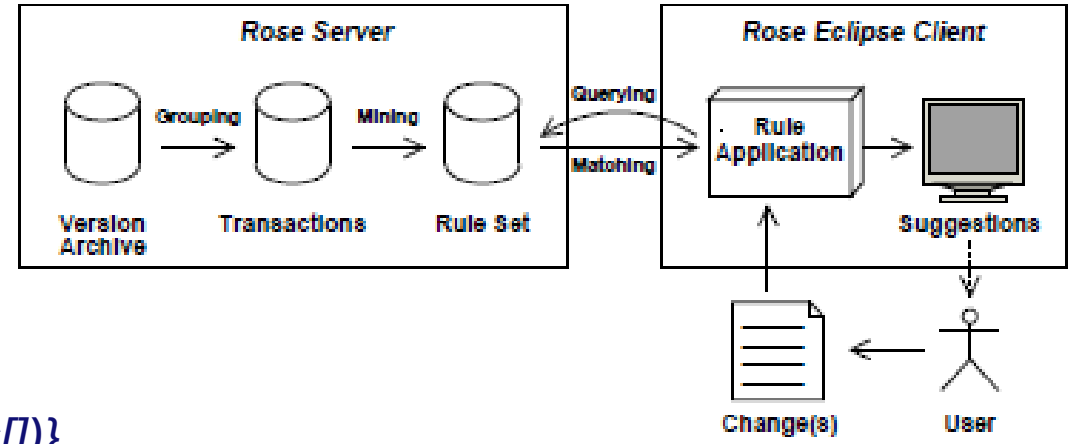

Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
*ComparePreferencePage.java
public final OverlayPreferenceStore.OverlayKey[] fKeys= new OverlayPreferenceStore.OverlayK
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, OPEN_STRUCTURE_COM
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SYNCHRONIZE_SCROLL
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_PSEUDO_CONFLI
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, INITIALLY_SHOW_ANC
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_MORE_INFO),
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, IGNORE_WHITESPACE)
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, PREF_SAVE_ALL_EDIT
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, NEW_PREFERENCE),
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.STRING, AbstractTextEditor.
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, AbstractTextEditor
//new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SPLINES),
new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SINGLE_LINE),
//new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_RESOLVE_UI),
};
public static void initDefaults(IPreferenceStore store) {
store.setDefault(OPEN_STRUCTURE_COMPARE, true);
store.setDefault(SYNCHRONIZE_SCROLLING, true);
store.setDefault(SHOW_PSEUDO_CONFLICTS, false);
store.setDefault(INITIALLY_SHOW_ANCESTOR_PANE, false);
store.setDefault(SHOW_MORE_INFO, false);
store.setDefault(IGNORE_WHITESPACE, false);
store.setDefault(PREF_SAVE_ALL_EDITORS, false);
//store.setDefault(USE_SPLINES, false);
store.setDefault(USE_SINGLE_LINE, true);
A) The user inserts a new preference into the field fKeys[]
B) ROSE suggests locations for further changes, e.g. the function initDefaults()
ROSE keeps a list of association rules
Related Changes
Symbol
File
Support
Confidence
initDefaults(IPreferenceStore store)
ComparePreferencePage.java
8
1.0
org.eclipse.compare/plugin.properties
plugin.properties
7
0.875
org.eclipse.compare/buildnotes_compare.html
buildnotes_compare.html
6
0.75
TextMergeViewer(Composite parent, int style, CompareConfiguration configuration)
TextMergeViewer.java
6
0.75
propertyChange(PropertyChangeEvent event)
TextMergeViewer.java
6
0.75
createGeneralPage(Composite parent)
ComparePreferencePage.java
5
0.625
createTextComparePage(Composite parent)
ComparePreferencePage.java
5
0.625
handleDispose(DisposeEvent event)
TextMergeViewer.java
4
0.5
Tasks Console Related Changes

eROSE



- **ROSE alerts for incomplete changes**

How?



Association rules:

{(Comp.java, *field, fKeys[])}*

⇒ { (Comp.java, *method, initDefaults()),*

(plug.properties, *file, plug.properties) }*

Experimental evaluation

- **Recall: 0.15**
  - **suggestion included 15% of all changes that were carried out**
- **Precision: 0.26**
  - **26% of all recommendations were correct**
- **Likelyhood:**
  - **70% of all transactions, topmost three suggestions contain a changed entity.**
- **EROSE learns quickly**
  - **within 30 days**
- **Extensive evaluation**

Conclusions

- Code cloning
- Differencing
  - Two approaches to identification of related differences:
    - Both based on data mining/rule learning
      - [Kim and Notkin] FOL rules
      - [EROSE] Association rules
    - Interesting ideas, not always impressive results
      - A lot of improvement is possible!