

1 Early Concepts

- **Nodes added by left/right:**

$$A_L \leftarrow \{l \in L \mid (\neg \exists b \in B)(l.id = b.id)\}$$

$$A_R \leftarrow \{r \in R \mid (\neg \exists b \in B)(r.id = b.id)\}$$

- **Nodes deleted by left/right:**

$$D_L \leftarrow \{b \in B \mid (\neg \exists l \in L)(b.id = l.id)\}$$

$$D_R \leftarrow \{b \in B \mid (\neg \exists r \in R)(b.id = r.id)\}$$

- **Nodes edited by left/right:**

$$E_L \leftarrow \{l \in L \mid (\exists b \in B)(\exists r \in R)(l.id = b.id = r.id \wedge l.body \neq b.body \wedge b.body = r.body)\}$$

$$E_R \leftarrow \{r \in R \mid (\exists b \in B)(\exists l \in L)(r.id = b.id = l.id \wedge r.body \neq b.body \wedge b.body = l.body)\}$$

2 Semistructured Merge

2.1 Early Concepts

- Every node's origin is set to UNKNOWN beforehand

2.2 Merge Algorithms

Algorithm 1: Merge Files

```

Input: l, b, r, o
1 if  $l.content = b.content$  then
2   |  $o.content \leftarrow r.content$ ;
3 else if  $b.content = r.content \vee l.content = r.content$  then
4   |  $o.content \leftarrow l.content$ ;
5 else
6   |  $L \leftarrow fileToTree(l)$ ;
7   |  $B \leftarrow fileToTree(b)$ ;
8   |  $R \leftarrow fileToTree(r)$ ;
9   |  $M \leftarrow mergeTrees(L, B, R)$ ;
10  |  $H \leftarrow getActiveHandlers()$ ;
11  | foreach  $h \in H$  do
12  |   |  $h.handle(M)$ ;
13  | end
14  |  $o.content \leftarrow treeToText(M)$ ;
15 end

```

Algorithm 2: Merge Trees

```

Input: L, B, R
Output: result of merging left, base and right trees
1  $L.origin = LEFT$ ;
2  $B.origin = BASE$ ;
3  $R.origin = RIGHT$ ;
4  $LB \leftarrow mergeNodes(L, B)$ ;
5  $M \leftarrow mergeNodes(LB, R)$ ;
6 foreach  $d \in D_L \cap D_R$  do
7   |  $removeNode(d, M)$ ;
8 end
9  $runTextualMergeOnLeaves(M)$ ;
10 return  $M$ ;

```

Algorithm 3: Run Textual Merge On Leaves**Input:** T

```

1 foreach  $t \in T.children$  do
2   |  $runTextualMergeOnLeaves(t)$ ;
3 end
4 if  $T.children = \emptyset \wedge SEPARATOR \in T.body$  then
5   |  $l, b, r \leftarrow split(T.body, SEPARATOR)$ ;
6   |  $l \leftarrow l - MARKER$ ;
7   |  $T.body \leftarrow textualMerge(l, b, r)$ ;
8 end

```

Algorithm 4: Merge Nodes**Input:** A, B **Output:** result of merging nodes A and B

```

1 if  $A = null$  then return  $B$ ;
2 if  $B = null$  then return  $A$ ;
3 if  $A.type \neq B.type \vee A.id \neq B.id$  then return  $null$ ;
4  $M.id \leftarrow B.id$ ;
5  $M.type \leftarrow B.type$ ;
6  $M.origin \leftarrow B.origin$ ;
7  $M.children \leftarrow \emptyset$ ;
8 if  $A.children = \emptyset \wedge B.children = \emptyset$  then
9   | if  $MARKER \in A.body$  then
10    |  $M.body \leftarrow A.body + B.body$ ;
11   | else if  $A.origin = LEFT \wedge B.origin = BASE$  then
12    |  $M.body \leftarrow MARKER + A.body + SEPARATOR + B.body + SEPARATOR$ ;
13   | else if  $A.origin = LEFT$  then
14    |  $M.body \leftarrow MARKER + A.body + SEPARATOR + SEPARATOR + B.body$ ;
15   | else
16    |  $M.body \leftarrow MARKER + SEPARATOR + A.body + SEPARATOR + B.body$ ;
17   | end
18   | return  $M$ ;
19 else if  $A.children \neq \emptyset \wedge B.children \neq \emptyset$  then
20   | foreach  $b \in B.children$  do
21     |  $a \leftarrow find(a \in A.children \rightarrow a.type = b.type \wedge a.id = b.id)$ ;
22     | if  $a.origin = UNKNOWN$  then  $a.origin \leftarrow A.origin$ ;
23     | if  $b.origin = UNKNOWN$  then  $b.origin \leftarrow B.origin$ ;
24     |  $M.children \leftarrow M.children \cup mergeNodes(a, b, step)$ ;
25   | end
26   | foreach  $a \in A.children$  do
27     |  $b \leftarrow find(b \in B.children \rightarrow a.type = b.type \wedge a.id = b.id)$ ;
28     | if  $a.origin = UNKNOWN$  then  $a.origin \leftarrow A.origin$ ;
29     | if  $b = null$  then  $M.children \leftarrow M.children \cup a$ ;
30   | end
31   | return  $M$ ;
32 end
33 return  $null$ ;

```

3 Handlers

3.1 Renaming Handler

3.1.1 Early Concepts

- Possibly renamed without body changes nodes:

$$R_{wobc}(T, B) = \{b \in B \mid (\neg \exists t \in T)(t.id = b.id) \wedge (\exists t \in T)(t.body = b.body)\}$$

- Possibly deleted or renamed with body changes nodes:

$$DR_{wbc}(T, B) = \{b \in B \mid (\neg \exists t \in T)(t.id = b.id \vee t.body = b.body)\}$$

- Nodes IDs similarity:

$$a.id \approx b.id \leftrightarrow a.id.name = b.id.name \vee a.id.params = b.id.params$$

3.1.2 Match Algorithm

Algorithm 5: Match Algorithm

Input: L, B, R, M

Output: Set of quadruples (l, b, r, m) consisting of the base node b and its corresponding left node l , right node r and merge node m

```

1 matches ← ∅;
2 foreach  $b \in DR_{wbc}(L, B) \cup DR_{wbc}(R, B) \cup R_{wobc}(L, B) \cup R_{wobc}(R, B)$  do
3    $l \leftarrow \text{correspondentNode}(b, L)$ ;
4    $r \leftarrow \text{correspondentNode}(b, R)$ ;
5    $m \leftarrow \text{mergeNode}(l, r, M)$ ;
6   matches ← matches  $\cup (l, b, r, m)$ ;
7 end
8 return matches

```

Algorithm 6: Correspondent Node

Input: b, T

Output: b's correspondent node on tree T

```

1  $t \leftarrow \text{findFirst}(t \in T \rightarrow t.id = b.id)$ ;
2 if  $t = \text{null}$  then
3    $t \leftarrow \text{findFirst}(t \in T \rightarrow t.body = b.body)$ ;
4 end
5 if  $t = \text{null}$  then
6    $t \leftarrow \text{findFirst}(t \in T \rightarrow t.body \approx b.body \wedge t.id \approx b.id)$ ;
7 end
8 if  $t = \text{null}$  then
9    $t \leftarrow \text{findFirst}(t \in T \rightarrow t.body = \text{substring}(b.body) \vee b.body = \text{substring}(t.body))$ ;
10 end
11 return t;

```

Algorithm 7: Merge Node

Input: l, r, M
Output: l and r 's merge node on tree M

```

1 if  $l \neq null$  then
2   | return  $find(m \in M \rightarrow m.id = l.id)$ ;
3 end
4 if  $r \neq null$  then
5   | return  $find(m \in M \rightarrow m.id = r.id)$ ;
6 end
7 return  $null$ ;

```

3.1.3 Handler Algorithms**Algorithm 8: Check References and Merge Methods Variant**

Input: $(l, b, r, m), M$

```

1 if  $l.id = b.id \vee r.id = b.id$  then
2   |  $m.body = textualMerge(l, b, r)$ ;
3   |  $removeUnmatchedNode(l, r, m, M)$ ;
4 else if  $l.id \neq r.id$  then
5   |  $m.body = conflict(l.body, b.body, r.body)$ ;
6   |  $removeUnmatchedNode(l, r, m, M)$ ;
7 else if  $l.body \neq r.body$  then
8   | if  $newReferenceTo(l) \vee newReferenceTo(r)$  then
9   |   |  $m.body = conflict(l.body, b.body, r.body)$ ;
10  | else
11  |   |  $m.body = textualMerge(l, b, r)$ ;
12  | end
13  |  $removeUnmatchedNode(l, r, m, M)$ ;
14 end

```

Algorithm 9: Merge Methods Variant

Input: $(l, b, r, m), M$

```

1  $m.body = textualMerge(l, b, r)$ ;
2  $removeUnmatchedNode(l, r, m, M)$ ;

```

Algorithm 10: Check Textual and Keep Both Methods Variant

Input: $(l, b, r, m), M$

```

1 if  $l.id = b.id \vee r.id = b.id$  then
2   | if  $textualMergeHasConflictInvolvingSignature(b)$  then
3   |   |  $m.body = conflict(l.body, b.body, r.body)$ ;
4   |   |  $removeUnmatchedNode(l, r, m, M)$ ;
5   | end
6 else if  $l.id \neq r.id \wedge l.body = r.body$  then
7   |  $m.body = conflict(l.body, b.body, r.body)$ ;
8   |  $removeUnmatchedNode(l, r, m, M)$ ;
9 end

```

Algorithm 11: Keep Both Methods Variant**Input:** $(l, b, r, m), M$

```

1 if  $(l.id = b.id \vee r.id = b.id) \wedge \text{hasConflict}(m)$  then
2   |  $\text{removeConflict}(m);$ 
3 end

```

Algorithm 12: Remove Unmatched Node**Input:** l, r, m, M

```

1 if  $l.id = m.id \wedge r.id \neq m.id$  then
2   |  $\text{removeNode}(r, M);$ 
3 end

```

3.2 Initialization Blocks Handler**3.2.1 Handler Algorithm****Algorithm 13: Handle****Input:** L, B, R, M

```

1  $IB_L \leftarrow \{n \in A_L \mid n.type = INITBLOCK\};$ 
2  $IB_R \leftarrow \{n \in A_R \mid n.type = INITBLOCK\};$ 
3  $IB_B \leftarrow \{n \in D_L \cap D_R \mid n.type = INITBLOCK\};$ 
4  $matches \leftarrow \emptyset;$ 
5 if  $|IB_L| = 1 \wedge |IB_B| = 1 \wedge |IB_R| = 1$  then
6   |  $matches \leftarrow matches \cup (IB_{L_1}, IB_{B_1}, IB_{R_1});$ 
7 else
8   foreach  $b \in IB_B$  do
9     |  $l \leftarrow \text{findFirst}(l \in IB_L \rightarrow l.body \approx b.body);$ 
10    |  $r \leftarrow \text{findFirst}(r \in IB_R \rightarrow r.body \approx b.body);$ 
11    |  $IB_L \leftarrow IB_L - l;$ 
12    |  $IB_R \leftarrow IB_R - r;$ 
13    | if  $l \neq null \wedge r \neq null$  then
14      |  $matches \leftarrow matches \cup (l, b, r);$ 
15    | end
16  end
17  foreach  $l \in IB_L$  do
18    |  $r \leftarrow \text{findFirst}(r \in IB_R \rightarrow r.body \approx l.body);$ 
19    |  $IB_R \leftarrow IB_R - r;$ 
20    | if  $r \neq null$  then
21      |  $matches \leftarrow matches \cup (l, null, r);$ 
22    | end
23  end
24 end
25 foreach  $(l, b, r) \in matches$  do
26   |  $m \leftarrow \text{find}(m \in M \rightarrow m.body = l.body);$ 
27   |  $m.body \leftarrow \text{textualMerge}(l.body, b.body, r.body);$ 
28   |  $m \leftarrow \text{find}(m \in M \rightarrow m.body = r.body);$ 
29   |  $\text{removeNode}(m, M);$ 
30 end

```

3.3 Multiple Initialization Blocks Handler

3.3.1 Handler Algorithm

Algorithm 14: Handle

```

Input: L, B, R, M
1   $IB_L \leftarrow \{n \in A_L \mid n.type = INITBLOCK\};$ 
2   $IB_R \leftarrow \{n \in A_R \mid n.type = INITBLOCK\};$ 
3   $IB_B \leftarrow \{n \in D_L \cap D_R \mid n.type = INITBLOCK\};$ 
4   $eIB_L \leftarrow \text{editedNodes}(IB_L, IB_B);$ 
5   $eIB_R \leftarrow \text{editedNodes}(IB_R, IB_B);$ 
6   $dIB_L \leftarrow \text{deletedNodes}(IB_L, IB_B, eIB_L);$ 
7   $dIB_R \leftarrow \text{deletedNodes}(IB_R, IB_B, eIB_R);$ 
8  foreach  $b \in IB_B$  do
9       $l \leftarrow eIB_L[b];$ 
10      $r \leftarrow eIB_R[b];$ 
11     if  $l \neq \text{null} \wedge r \neq \text{null}$  then
12          $\text{updateMergeTree}(l, b, r, M);$ 
13     else if  $l \neq \text{null} \vee r \neq \text{null}$  then
14         if  $l \neq \text{null}$  then
15              $r \leftarrow \text{find}(r \in dIB_R \rightarrow r.body = b.body);$ 
16             if  $r \neq \text{null}$  then  $\text{removeNode}(b, M);$  ;
17         else
18              $l \leftarrow \text{find}(l \in dIB_L \rightarrow l.body = b.body);$ 
19             if  $l \neq \text{null}$  then  $\text{removeNode}(b, M);$  ;
20         end
21          $\text{updateMergeTree}(l, b, r, M);$ 
22     else
23          $m \leftarrow \text{find}(m \in M \rightarrow m.body = b.body);$ 
24          $\text{removeNode}(m, M);$ 
25     end
26 end
27  $aIB_L \leftarrow \text{addedNodes}(IB_L, IB_B, eIB_L);$ 
28  $aIB_R \leftarrow \text{addedNodes}(IB_R, IB_B, eIB_R);$ 
29  $DEP \leftarrow \text{dependentNodes}(aIB_L, aIB_R);$ 
30 foreach  $(l, rs) \in DEP$  do
31      $s \leftarrow \varepsilon;$ 
32     foreach  $r \in rs$  do
33          $s \leftarrow s + r.body;$ 
34          $m \leftarrow \text{find}(m \in M \rightarrow m.body = r.body);$ 
35          $\text{removeNode}(r, M);$ 
36     end
37      $m \leftarrow \text{find}(m \in M \rightarrow m.body = l.body);$ 
38      $m.body \leftarrow \text{conflict}(l.body, \varepsilon, s);$ 
39 end
40 foreach  $l \in aIB_L$  do
41     foreach  $r \in aIB_R$  do
42         if  $l.body = r.body$  then
43              $m \leftarrow \text{find}(m \in M \rightarrow m.body = r.body);$ 
44              $\text{removeNode}(m, M);$ 
45         end
46     end
47 end

```

Algorithm 15: Edited Nodes**Input:** IB, IB_B **Output:** map associating a deleted base node b in IB_B and its correspondent added branch node a in IB

```

1  $D \leftarrow \{d \in IB_B \mid (\neg \exists a \in IB)(d.body = a.body)\};$ 
2  $A \leftarrow \{a \in IB \mid (\neg \exists d \in IB_B)(a.body = d.body)\};$ 
3  $matches \leftarrow \emptyset;$ 
4 foreach  $a \in A$  do
5    $S \leftarrow \{d \in D \mid a.body \approx d.body\};$ 
6    $b \leftarrow \underset{s \in S}{\operatorname{argmax}} (\operatorname{similarity}(s.body, a.body));$ 
7   if  $b \neq null$  then  $matches \leftarrow matches \cup \{b : a\};$ 
8 end
9 return  $matches$ 

```

Algorithm 16: Added Nodes**Input:** IB, IB_B, eIB **Output:** set of initialization block nodes added by branch

```

1  $A \leftarrow \{n \in IB \mid (\neg \exists b \in IB_B)(n.body = b.body)\};$ 
2  $A \leftarrow \{n \in A \mid (\neg \exists e \in eIB)(n.body = e.value.body)\};$ 
3 return  $A;$ 

```

Algorithm 17: Deleted Nodes**Input:** IB, IB_B, eIB **Output:** set of initialization block nodes deleted by branch

```

1  $D \leftarrow \{b \in IB_B \mid (\neg \exists n \in IB)(b.body = n.body)\};$ 
2  $D \leftarrow \{n \in D \mid (\neg \exists e \in eIB)(n.body = e.key.body)\};$ 
3 return  $D;$ 

```

Algorithm 18: Update Merge Tree**Input:** l, b, r, M

```

1  $m \leftarrow \operatorname{find}(m \in M \rightarrow m.body = l.body);$ 
2  $m.body \leftarrow \operatorname{textualMerge}(l.body, b.body, r.body);$ 
3  $m \leftarrow \operatorname{find}(m \in M \rightarrow m.body = r.body);$ 
4 removeNode $(m, M);$ 

```

Algorithm 19: Dependent Nodes**Input:** aIB_L, aIB_R **Output:** map associating an added left node l in aIB_L and all added right nodes r in aIB_R with common global variables

```

1  $DEP \leftarrow \emptyset;$ 
2 foreach  $l \in aIB_L$  do
3    $DEP \leftarrow DEP \cup \{l : \emptyset\};$ 
4    $V_L \leftarrow \operatorname{globalVariables}(l);$ 
5   foreach  $r \in aIB_R$  do
6      $V_R \leftarrow \operatorname{globalVariables}(r);$ 
7     if  $V_L \cap V_R \neq \emptyset$  then  $DEP[l] \leftarrow DEP[l] \cup r;$ 
8   end
9 end
10 return  $DEP;$ 

```

3.4 Type Ambiguity Error Handler

3.4.1 Handler Algorithm

Algorithm 20: Handle

```

Input: L, B, R, M
1   $ID_L \leftarrow \{n \in A_L \mid n.type = IMPORTDECL\};$ 
2   $ID_R \leftarrow \{n \in A_R \mid n.type = IMPORTDECL\};$ 
3  if  $ID_L = \emptyset \vee ID_R = \emptyset$  then return;
4   $M_U \leftarrow \text{textualMerge}(\text{treeToText}(L), \text{treeToText}(B), \text{treeToText}(R));$ 
5   $I_L, I_R \leftarrow \text{extractInsertions}(M_U);$ 
6   $cs \leftarrow \text{extractConflicts}(M_U);$ 
7   $c \leftarrow \text{compile}(M_U);$ 
8   $ps \leftarrow \text{problems}(c);$ 
9  foreach  $l \in ID_L$  do
10 |    $m_l \leftarrow \text{extractPackageMember}(l.body);$ 
11 |   foreach  $r \in ID_R$  do
12 | |    $m_r \leftarrow \text{extractPackageMember}(r.body);$ 
13 | |   if  $m_l = m_r$  then
14 | | |    $p \leftarrow \text{importDeclarationsProblem}(l, r, ps);$ 
15 | | |   if  $p \neq \text{null}$  then
16 | | | |    $m \leftarrow \text{find}(m \in M \rightarrow m.body = l.body);$ 
17 | | | |    $m.body \leftarrow \text{conflict}(l.body, \varepsilon, r.body);$ 
18 | | | |    $m \leftarrow \text{find}(m \in M \rightarrow m.body = r.body);$ 
19 | | | |    $\text{removeNode}(m, M);$ 
20 | | | |    $ps \leftarrow ps - p;$ 
21 | | | |   break;
22 | |   end
23 | |   else if  $(m_l = * \vee m_r = *) \wedge \text{importDeclarationsConflict}(l, r, cs)$  then
24 | | |    $I \leftarrow I_L;$ 
25 | | |    $m \leftarrow m_r;$ 
26 | | |   if  $m_l \neq *$  then
27 | | | |    $I \leftarrow I_R;$ 
28 | | | |    $m \leftarrow m_l;$ 
29 | | |   end
30 | | |    $i \leftarrow \text{find}(i \in I \rightarrow \text{import} \notin i \wedge m \in i);$ 
31 | | |   if  $i \neq \text{null}$  then
32 | | | |    $m \leftarrow \text{find}(m \in M \rightarrow m.body = l.body);$ 
33 | | | |    $m.body \leftarrow \text{conflict}(l.body, \varepsilon, r.body);$ 
34 | | | |    $m \leftarrow \text{find}(m \in M \rightarrow m.body = r.body);$ 
35 | | | |    $\text{removeNode}(m, M);$ 
36 | | | |   break;
37 | | |   end
38 | |   end
39 |   end
40 end

```


Algorithm 21: Import Declarations Problem**Input:** l, r, ps **Output:** compilation problem in ps concerning l and r import declarations, if there is one

```

1 foreach  $p \in ps$  do
2   if  $p.type = COLLISION$  then
3     foreach  $a \in p.arguments$  do
4       if  $a \in l.body \vee a \in r.body$  then return  $p$ ;
5     end
6   else if  $p.type = AMBIGUITY$  then return  $p$ ;
7 end
8 return  $null$ ;

```

Algorithm 22: Import Declarations Conflict**Input:** l, r, cs **Output:** whether there is a unstructured conflict in cs concerning l and r import declarations

```

1 foreach  $c \in cs$  do
2   if  $l.body \in c.left \wedge r.body \in c.right$  then return  $TRUE$ ;
3 end
4 return  $FALSE$ ;

```

3.5 New Element Referencing Edited One Handler

3.5.1 Handler Algorithm

Algorithm 23: Handle

Input: L, B, R, M

```

1  $M_U \leftarrow \text{textualMerge}(\text{treeToText}(L), \text{treeToText}(B), \text{treeToText}(R));$ 
2  $cs \leftarrow \text{extractConflicts}(M_U);$ 
3  $aMFD_L \leftarrow \{l \in A_L \mid l.type = METHODDECL \vee l.type = FIELDDECL\};$ 
4  $aMFD_R \leftarrow \{r \in A_R \mid r.type = METHODDECL \vee r.type = FIELDDECL\};$ 
5  $eMFD_L \leftarrow \{l \in E_L \mid l.type = METHODDECL \vee l.type = FIELDDECL\};$ 
6  $eMFD_R \leftarrow \{r \in E_R \mid r.type = METHODDECL \vee r.type = FIELDDECL\};$ 
7 foreach  $a_l \in aMFD_L$  do
8   foreach  $e_r \in eMFD_R$  do
9     if  $\text{nodesConflict}(a_l, e_r, cs) \wedge e_r.id.name \in a_l.body$  then
10        $b \leftarrow \text{find}(b \in B \rightarrow b.id = e_r.id);$ 
11        $m \leftarrow \text{find}(m \in M \rightarrow m.body = e_r.body);$ 
12        $m.body \leftarrow \text{conflict}(e_r.body, b.body, a_l.body);$ 
13        $m \leftarrow \text{find}(m \in M \rightarrow m.body = a_l.body);$ 
14        $\text{removeNode}(m, M);$ 
15     end
16   end
17 end
18 foreach  $a_r \in aMFD_R$  do
19   foreach  $e_l \in eMFD_L$  do
20     if  $\text{nodesConflict}(a_r, e_l, cs) \wedge r_l.id.name \in a_r.body$  then
21        $b \leftarrow \text{find}(b \in B \rightarrow b.id = e_l.id);$ 
22        $m \leftarrow \text{find}(m \in M \rightarrow m.body = e_l.body);$ 
23        $m.body \leftarrow \text{conflict}(e_l.body, b.body, a_r.body);$ 
24        $m \leftarrow \text{find}(m \in M \rightarrow m.body = a_r.body);$ 
25        $\text{removeNode}(m, M);$ 
26     end
27   end
28 end

```

Algorithm 24: Nodes Conflict

Input: a, b, cs

Output: whether there is an unstructured conflict in cs concerning a and b nodes

```

1 foreach  $c \in cs$  do
2   if  $c.left = a.body \wedge c.right = b.body$  then return  $TRUE$ ;
3   if  $c.left = b.body \wedge c.right = a.body$  then return  $TRUE$ ;
4 end
5 return  $FALSE$ ;

```