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 ← r.content;
3  else if b.content = r.content ∨ l.content = r.content then
4  |  o.content ← l.content;
5  else
6  |  L ← fileToTree(l);
7  |  B ← fileToTree(b);
8  |  R ← fileToTree(r);
9  |  M ← mergeTrees(L, B, R);
10 |  H ← getActiveHandlers();
11 |  foreach h ∈ H do
12 |  |  h.handle(M);
13 |  end
14 |  o.content ← 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 ← mergeNodes(L, B);
5  M ← mergeNodes(LB, R);
6  foreach d ∈ D_L ∩ D_R do
7  |  removeNode(d, M);
8  end
9  runTextualMergeOnLeaves(M);
10 return M;
```

Algorithm 3: Run Textual Merge On Leaves

Input: T

foreach $t \in T.children$ **do**
  `runTextualMergeOnLeaves(`t`)`;
end
if $T.children = \emptyset \wedge SEPARATOR \in T.body$ **then**
  $l, b, r \leftarrow split(T.body, SEPARATOR)$;
  $l \leftarrow l - MARKER$;
  $T.body \leftarrow$ `textualMerge(`l, b, r`)`;
end

Algorithm 4: Merge Nodes

Input: A, B
Output: result of merging nodes A and B

if $A =$ **null then return** B;
if $B =$ **null then return** A;
if $A.type \neq B.type \vee A.id \neq B.id$ **then return null**;
$M.id \leftarrow B.id$;
$M.type \leftarrow B.type$;
$M.origin \leftarrow B.origin$;
$M.children \leftarrow \emptyset$;
if $A.children = \emptyset \wedge B.children = \emptyset$ **then**
  if $MARKER \in A.body$ **then**
    $M.body \leftarrow A.body + B.body$;
  else if $A.origin = LEFT \wedge B.origin = BASE$ **then**
    $M.body \leftarrow MARKER + A.body + SEPARATOR + B.body + SEPARATOR$;
  else if $A.origin = LEFT$ **then**
    $M.body \leftarrow MARKER + A.body + SEPARATOR + SEPARATOR + B.body$;
  else
    $M.body \leftarrow MARKER + SEPARATOR + A.body + SEPARATOR + B.body$;
  end
  return M;
else if $A.children \neq \emptyset \wedge B.children \neq \emptyset$ **then**
  foreach $b \in B.children$ **do**
    $a \leftarrow$ `find(`$a \in A.children \rightarrow a.type = b.type \wedge a.id = b.id$`)`;
    if $a.origin = UNKNOWN$ **then** $a.origin \leftarrow A.origin$;
    if $b.origin = UNKNOWN$ **then** $b.origin \leftarrow B.origin$;
    $M.children \leftarrow M.children \cup$ `mergeNodes(`a, b`)`;
  end
  foreach $a \in A.children$ **do**
    $b \leftarrow$ `find(`$b \in B.children \rightarrow a.type = b.type \wedge a.id = b.id$`)`;
    if $a.origin = UNKNOWN$ **then** $a.origin \leftarrow A.origin$;
    if $b =$ **null then** $M.children \leftarrow M.children \cup a$;
  end
  return M;
end
return null;

3 Handlers

3.1 Renaming Handler

3.1.1 Early Concepts

- **Nodes possibly renamed by left/right without body changes**:
  $R_L = \{b \in B \mid (\neg\exists l \in L)(b.id = l.id) \wedge (\exists l \in L)(b.body - l.body)\}$
  $R_R = \{b \in B \mid (\neg\exists r \in R)(b.id = r.id) \wedge (\exists r \in R)(b.body - r.body)\}$
- **Nodes possibly deleted or renamed by left/right with body changes**:
  $DR_L \leftarrow \{b \in B \mid (\neg\exists l \in L)(b.id = l.id \vee b.body = l.body)\}$
  $DR_R \leftarrow \{b \in B \mid (\neg\exists r \in R)(b.id = r.id \vee b.body = r.body)\}$
- **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 ∈ R_L ∪ R_R ∪ DR_L ∪ DR_R do
3 |   l ← correspondentNode(b, L);
4 |   r ← correspondentNode(b, R);
5 |   m ← mergeNode(l, r, M);
6 |   matches ← matches ∪ (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 ← findFirst(t ∈ T → t.id = b.id);
2 if t = null then
3 |   t ← findFirst(t ∈ T → t.body = b.body);
4 end
5 if t = null then
6 |   t ← findFirst(t ∈ T → t.id ≈ b.id ∧ t.body ≈ b.body);
7 end
8 if t = null then
9 |   t ← findFirst(t ∈ T → t.body = substring(b.body) ∨ b.body = 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 ≠ null then
2 |   return find(m ∈ M → m.id = l.id);
3 end
4 if r ≠ null then
5 |   return find(m ∈ M → 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 ∨ r.id = b.id then
2  |   m.body ← textualMerge(l, b, r);
3  |   removeUnmatchedNode(l, r, m, M);
4  else if l.id ≠ r.id then
5  |   m.body ← conflict(l.body, b.body, r.body);
6  |   removeUnmatchedNode(l, r, m, M);
7  else if l.body ≠ r.body then
8  |   if newReferenceTo(l) ∨ 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 ∨ 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 ≠ r.id ∧ 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 ∨ r.id = b.id) ∧ hasConflict(m) then
2 |   removeConflict(m);
3 end
```

Algorithm 12: Remove Unmatched Node

Input: l, r, m, M

```
1  if l.id = m.id ∧ r.id ≠ m.id then
2  |  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 ← {n ∈ A_L | n.type = INITBLOCK};
2  IB_R ← {n ∈ A_R | n.type = INITBLOCK};
3  IB_B ← {n ∈ D_L ∩ D_R | n.type = INITBLOCK};
4  matches ← ∅;
5  if |IB_L| = 1 ∧ |IB_B| = 1 ∧ |IB_R| = 1 then
6  |  matches ← matches ∪ (IB_L1, IB_B1, IB_R1);
7  else
8  |  foreach b ∈ IB_B do
9  |  |  l ← findFirst(l ∈ IB_L → l.body ≈ b.body);
10 |  |  r ← findFirst(r ∈ IB_R → r.body ≈ b.body);
11 |  |  IB_L ← IB_L − l;
12 |  |  IB_R ← IB_R − r;
13 |  |  if l ≠ null ∧ r ≠ null then
14 |  |  |  matches ← matches ∪ (l, b, r);
15 |  |  end
16 |  end
17 |  foreach l ∈ IB_L do
18 |  |  r ← findFirst(r ∈ IB_R → r.body ≈ l.body);
19 |  |  IB_R ← IB_R − r;
20 |  |  if r ≠ null then
21 |  |  |  matches ← matches ∪ (l, null, r);
22 |  |  end
23 |  end
24 end
25 foreach (l, b, r) ∈ matches do
26 |  m ← find(m ∈ M → m.body = l.body);
27 |  m.body ← textualMerge(l.body, b.body, r.body);
28 |  m ← find(m ∈ M → m.body = r.body);
29 |  removeNode(m, M);
30 end
```

3.3 Multiple Initialization Blocks Handler

3.3.1 Handler Algorithm

Algorithm 14: Handle

Input: L, B, R, M

$IB_L \leftarrow \{n \in A_L \mid n.type = INITBLOCK\}$;
$IB_R \leftarrow \{n \in A_R \mid n.type = INITBLOCK\}$;
$IB_B \leftarrow \{n \in D_L \cap D_R \mid n.type = INITBLOCK\}$;

$eIB_L \leftarrow$ `editedNodes`(IB_L, IB_B);
$eIB_R \leftarrow$ `editedNodes`(IB_R, IB_B);

$dIB_L \leftarrow$ `deletedNodes`(IB_L, IB_B, eIB_L);
$dIB_R \leftarrow$ `deletedNodes`(IB_R, IB_B, eIB_R);

foreach $b \in IB_B$ **do**
    $l \leftarrow eIB_L[b]$;
    $r \leftarrow eIB_R[b]$;
    if $l \neq$ **null** $\wedge\, r \neq$ **null then**
        `updateMergeTree`(l, b, r, M);
    else if $l \neq$ **null** $\vee\, r \neq$ **null then**
        if $l \neq$ **null then**
            $r \leftarrow$ `find`$(r \in dIB_R \rightarrow r.body = b.body)$;
            if $r \neq$ **null then** `removeNode`(b, M);
        else
            $l \leftarrow$ `find`$(l \in dIB_L \rightarrow l.body = b.body)$;
            if $l \neq$ **null then** `removeNode`(b, M);
        end
        `updateMergeTree`(l, b, r, M);
    else
        $m \leftarrow$ `find`$(m \in M \rightarrow m.body = b.body)$;
        `removeNode`(m, M);
    end
end

$aIB_L \leftarrow$ `addedNodes`(IB_L, IB_B, eIB_L);
$aIB_R \leftarrow$ `addedNodes`(IB_R, IB_B, eIB_R);

$DEP \leftarrow$ `dependentNodes`(aIB_L, aIB_R);
foreach $(l, rs) \in DEP$ **do**
    $s \leftarrow \varepsilon$;
    foreach $r \in rs$ **do**
        $s \leftarrow s + r.body$;
        $m \leftarrow$ `find`$(m \in M \rightarrow m.body = r.body)$;
        `removeNode`(r, M);
    end
    $m \leftarrow$ `find`$(m \in M \rightarrow m.body = l.body)$;
    $m.body \leftarrow$ `conflict`$(l.body, \varepsilon, s)$;
end

foreach $l \in aIB_L$ **do**
    foreach $r \in aIB_R$ **do**
        if $l.body = r.body$ **then**
            $m \leftarrow$ `find`$(m \in M \rightarrow m.body = r.body)$;
            `removeNode`(m, M);
        end
    end
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

$D \leftarrow \{d \in IB_B \mid (\neg\exists a \in IB)(d.body = a.body)\}$;
$A \leftarrow \{a \in IB \mid (\neg\exists d \in IB_B)(a.body = d.body)\}$;
$matches \leftarrow \emptyset$;
foreach $a \in A$ **do**
  $S \leftarrow \{d \in D \mid a.body \approx d.body\}$;
  $b \leftarrow \underset{s \in S}{\text{argmax}}$ (`similarity(`$s.body,\ a.body$`)`);
  if $b \neq$ **null then** $matches \leftarrow matches \cup \{b : a\}$;
end
return $matches$

Algorithm 16: Added Nodes

Input: IB, IB_B, eIB
Output: set of initilization block nodes added by branch

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

Algorithm 17: Deleted Nodes

Input: IB, IB_B, eIB
Output: set of initialization block nodes deleted by branch

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

Algorithm 18: Update Merge Tree

Input: l, b, r, M

$m \leftarrow$ `find(`$m \in M \rightarrow m.body = l.body$`)`;
$m.body \leftarrow$ `textualMerge(`$l.body,\ b.body,\ r.body$`)`;
$m \leftarrow$ `find(`$m \in M \rightarrow m.body = r.body$`)`;
`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

$DEP \leftarrow \emptyset$;
foreach $l \in aIB_L$ **do**
  $DEP \leftarrow DEP \cup \{l : \emptyset\}$;
  $V_L \leftarrow$ `globalVariables(`l`)`;
  foreach $r \in aIB_R$ **do**
    $V_R \leftarrow$ `globalVariables(`r`)`;
    if $V_L \cap V_R \neq \emptyset$ **then** $DEP[l] \leftarrow DEP[l] \cup r$;
  end
end
return DEP;

3.4 Type Ambiguity Error Handler

3.4.1 Handler Algorithm

Algorithm 20: Handle

Input: L, B, R, M

```
1  ID_L ← {n ∈ A_L | n.type = IMPORTDECL};
2  ID_R ← {n ∈ A_R | n.type = IMPORTDECL};
3  if ID_L = ∅ ∨ ID_R = ∅ then return;
4  T_L ← treeToText(L);
5  T_B ← treeToText(B);
6  T_R ← treeToText(R);
7  M_U ← textualMerge(T_L, T_B, T_R);
8  I_L, I_R ← extractInsertions(M_U);
9  cs ← extractConflicts(M_U);
10 c ← compile(M_U);
11 ps ← problems(c);
12 foreach l ∈ ID_L do
13     m_l ← extractPackageMember(l.body);
14     foreach r ∈ ID_R do
15         m_r ← extractPackageMember(r.body);
16         if m_l = m_r then
17             p ← importDeclarationsProblem(l, r, ps);
18             if p ≠ null then
19                 m ← find(m ∈ M → m.body = l.body);
20                 m.body ← conflict(l.body, ε, r.body);
21                 m ← find(m ∈ M → m.body = r.body);
22                 removeNode(m, M);
23                 ps ← ps − p;
24                 break;
25             end
26         else if (m_l = ∗ ∨ m_r = ∗) ∧ importDeclarationsConflict(l, r, cs) then
27             if m_l ≠ ∗ then
28                 I ← I_R;
29                 m ← m_l;
30             else
31                 I ← I_L;
32                 m ← m_r;
33             end
34             i ← find(i ∈ I → IMPORT ∉ i ∧ m ∈ i);
35             if i ≠ null then
36                 m ← find(m ∈ M → m.body = l.body);
37                 m.body ← conflict(l.body, ε, r.body);
38                 m ← find(m ∈ M → m.body = r.body);
39                 removeNode(m, M);
40                 break;
41             end
42         end
43     end
44 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 ∈ ps do
2 |   if p.type = COLLISION then
3 |   |   foreach a ∈ p.arguments do
4 |   |   |   if a ∈ l.body ∨ a ∈ 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: wether there is an unstructured conflict in cs concerning l and r import declarations

```
1 foreach c ∈ cs do
2 |   if l.body ∈ c.left ∧ r.body ∈ 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

$T_L \leftarrow$ `treeToText(`L`)`;
$T_B \leftarrow$ `treeToText(`B`)`;
$T_R \leftarrow$ `treeToText(`R`)`;
$M_U \leftarrow$ `textualMerge(`T_L, T_B, T_R`)`;
$cs \leftarrow$ `extractConflicts(`M_U`)`;
$aMFD_L \leftarrow \{l \in A_L \mid l.type = METHODDECL \vee l.type = FIELDDECL\}$;
$aMFD_R \leftarrow \{r \in A_R \mid r.type = METHODDECL \vee r.type = FIELDDECL\}$;
$eMFD_L \leftarrow \{l \in E_L \mid l.type = METHODDECL \vee l.type = FIELDDECL\}$;
$eMFD_R \leftarrow \{r \in E_R \mid r.type = METHODDECL \vee r.type = FIELDDECL\}$;
foreach $a_l \in aMFD_L$ **do**
  foreach $e_r \in eMFD_R$ **do**
    if `nodesConflict(`a_l, e_r, cs`)` $\wedge\ e_r.id.name \in a_l.body$ **then**
      $b \leftarrow$ `find(`$b \in B \rightarrow b.id = e_r.id$`)`;
      $m \leftarrow$ `find(`$m \in M \rightarrow m.body = e_r.body$`)`;
      $m.body \leftarrow$ `conflict(`$e_r.body$, $b.body$, $a_l.body$`)`;
      $m \leftarrow$ `find(`$m \in M \rightarrow m.body = a_l.body$`)`;
      `removeNode(`m, M`)`;
    end
  end
end
foreach $a_r \in aMFD_R$ **do**
  foreach $e_l \in eMFD_L$ **do**
    if `nodesConflict(`a_r, e_l, cs`)` $\wedge\ e_l.id.name \in a_r.body$ **then**
      $b \leftarrow$ `find(`$b \in B \rightarrow b.id = e_l.id$`)`;
      $m \leftarrow$ `find(`$m \in M \rightarrow m.body = e_l.body$`)`;
      $m.body \leftarrow$ `conflict(`$e_l.body$, $b.body$, $a_r.body$`)`;
      $m \leftarrow$ `find(`$m \in M \rightarrow m.body = a_r.body$`)`;
      `removeNode(`m, M`)`;
    end
  end
end

Algorithm 24: Nodes Conflict

Input: a, b, cs
Output: wether there is an unstructured conflict in cs concerning a and b nodes

foreach $c \in cs$ **do**
  if $c.left = a.body \wedge c.right = b.body$ **then return true**;
  if $c.left = b.body \wedge c.right = a.body$ **then return true**;
end
return false;

3.6 Deletions Handler

3.6.1 Handler Algorithm

Algorithm 25: Handle

Input: L, B, R, M

```
1  T_L ← treeToText(L);
2  T_B ← treeToText(B);
3  T_R ← treeToText(R);
4  foreach d_l ∈ D_L do
5      if d_l.children ≠ ∅ then
6          r ← find(r ∈ R → r.id = d_l.id);
7          m ← find(m ∈ M → m.id = d_l.id);
8          if sameShape(d_l, r) ∧ d_l.body = r.body then removeNode(m, M);
9          else if newReference(d_l.id, T_B, T_R) then m.parent.addChild(r, m.index);
10         else
11             removeNode(m, M);
12             a_l ← renamingMatch(A_L, d_l, T_B, T_L);
13             if a_l ≠ null then
14                 r.id ← a_l.id;
15                 removeNode(a_l, M);
16                 m.parent.addChild(r, m.index);
17             else
18                 n.id ← r.id;
19                 n.type ← r.type;
20                 n.body ← conflict(ε, d_l, r);
21                 m.parent.addChild(n, m.index);
22             end
23         end
24     end
25 end
26 foreach d_r ∈ D_R do
27     if d_r.children ≠ ∅ then
28         l ← find(l ∈ L → l.id = d_r.id);
29         m ← find(m ∈ M → m.id = d_r.id);
30         if sameShape(d_r, l) ∧ d_r.body = l.body then removeNode(m, M);
31         else if newReference(d_r.id, T_B, T_L) then m.parent.addChild(l, m.index);
32         else
33             removeNode(m, M);
34             a_r ← renamingMatch(A_R, d_r, T_B, T_R);
35             if a_r ≠ null then
36                 l.id ← a_r.id;
37                 removeNode(a_r, M);
38                 m.parent.addChild(l, m.index);
39             else
40                 n.id ← l.id;
41                 n.type ← l.type;
42                 n.body ← conflict(l, d_r, ε);
43                 m.parent.addChild(n, m.index);
44             end
45         end
46     end
47 end
```

Algorithm 26: Same Shape

Input: A, B
Output: wether nodes A and B have same shape

```
1 if A.children = ∅ ∧ B.children = ∅ then return A.type = B.type;
2 if A.children = ∅ ∨ B.children = ∅ then return false;
3 if |A.children| ≠ |B.children| then return false;
4 result ← true;
5 foreach (a, b) ∈ (A.children, B.children) do
6 |   result ← result ∧ sameShape(a, b);
7 end
8 return result;
```

Algorithm 27: New Reference

Input: id, T_B, T
Output: wether there is a new reference to id in T

```
1 return countReferences(id, T) > countReferences(id, T_B);
```

Algorithm 28: Renaming Match

Input: A, d, T_B, T
Output: added node a in A with the same shape and similar body as deleted node d, such that there are no new references to a's id in T

```
1 return find(a ∈ A → sameShape(a, d) ∧ a.body ≈ d.body ∧ ¬newReference(a.id, T_B, T));
```