1 Semistructured Merge

1.1 Early Concepts

1. **Every node's origin is set to UNKNOWN beforehand**
2. **Nodes added by left**:
   $A_L \leftarrow \{l \in L \mid (\neg\exists b \in B)(l.id = b.id)\}$
3. **Nodes added by right**:
   $A_R \leftarrow \{r \in R \mid (\neg\exists b \in B)(r.id = b.id)\}$
4. **Nodes deleted from base**:
   $D_B \leftarrow \{b \in B \mid (\neg\exists l \in L)(b.id = l.id) \wedge (\neg\exists r \in R)(b.id = r.id)\}$

1.2 Merge Algorithms

Algorithm 1: Merge Files

Input: l, b, r, o

if $l.content = b.content$ **then**
  $o.content \leftarrow r.content$;
else if $b.content = r.content \vee l.content = r.content$ **then**
  $o.content \leftarrow l.content$;
else
  $L \leftarrow$ `fileToTree(`l`)`;
  $B \leftarrow$ `fileToTree(`b`)`;
  $R \leftarrow$ `fileToTree(`r`)`;
  $M \leftarrow$ `mergeTrees(`L, B, R`)`;
  $H \leftarrow$ `getActiveHandlers()`;
  foreach $h \in H$ **do**
    $h.handle(M)$;
  end
  $o.content \leftarrow$ `treeToText(`M`)`;
end

Algorithm 2: Merge Trees

Input: L, B, R
Output: result of merging left, base and right trees

$L.origin = LEFT$;
$B.origin = BASE$;
$R.origin = RIGHT$;
$LB \leftarrow$ `mergeNodes(`L, B`)`;
$M \leftarrow$ `mergeNodes(`LB, R`)`;
foreach $d \in D_B$ **do**
  `removeNode(`d, M`)`;
end
`runTextualMergeOnLeaves(`M`)`;
return M;

Algorithm 3: Run Textual Merge On Leaves

Input: T

```
foreach t ∈ T.children do
    runTextualMergeOnLeaves(t);
end
if T.children = ∅ ∧ SEPARATOR ∈ T.body then
    l, b, r ← split(T.body, SEPARATOR);
    l ← l − MARKER;
    T.body ← 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 ≠ B.type ∨ A.id ≠ B.id then return null;
M.id ← B.id;
M.type ← B.type;
M.origin ← B.origin;
M.children ← ∅;
if A.children = ∅ ∧ B.children = ∅ then
    if MARKER ∈ A.body then
        M.body ← A.body + B.body;
    else if A.origin = LEFT ∧ B.origin = BASE then
        M.body ← MARKER + A.body + SEPARATOR + B.body + SEPARATOR;
    else if A.origin = LEFT then
        M.body ← MARKER + A.body + SEPARATOR + SEPARATOR + B.body;
    else
        M.body ← MARKER + SEPARATOR + A.body + SEPARATOR + B.body;
    end
    return M;
else if A.children ≠ ∅ ∧ B.children ≠ ∅ then
    foreach b ∈ B.children do
        a ← find(a ∈ A.children → a.type = b.type ∧ a.id = b.id);
        if a.origin = UNKNOWN then a.origin ← A.origin;
        if b.origin = UNKNOWN then b.origin ← B.origin;
        M.children ← M.children ∪ mergeNodes(a, b, step);
    end
    foreach a ∈ A.children do
        b ← find(b ∈ B.children → a.type = b.type ∧ a.id = b.id);
        if a.origin = UNKNOWN then a.origin ← A.origin;
        if b = null then M.children ← M.children ∪ a;
    end
    return M;
end
return null;
```

2 Handlers

2.1 Renaming Handler

2.1.1 Early Concepts

1. **Possibly renamed without body changes nodes**:
   $R_{wobc}(T, B) = \{b \in B \mid (\neg\exists t \in T)(t.id = b.id)) \wedge (\exists t \in T)(t.body = b.body)\}$
2. **Possibly deleted or renamed with body changes nodes**:
   $DR_{wbc}(T, B) = \{b \in B \mid (\neg\exists t \in T)(t.id = b.id \vee t.body = b.body)\}$
3. **Nodes IDs similarity**:
   $a.id \approx b.id \leftrightarrow a.id.name = b.id.name \vee a.id.params = b.id.params$

2.1.2 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

$matches \leftarrow \emptyset$;
foreach $b \in DR_{wbc}(L, B) \cup DR_{wbc}(R, B) \cup R_{wobc}(L, B) \cup R_{wobc}(R, B)$ **do**
  $l \leftarrow$ `correspondentNode(`b, L`)`;
  $r \leftarrow$ `correspondentNode(`b, R`)`;
  $m \leftarrow$ `mergeNode(`l, r, M`)`;
  $matches \leftarrow matches \cup (l, b, r, m)$;
end
return $matches$

Algorithm 6: Correspondent Node

Input: b, T
Output: b's correspondent node on tree T

$t \leftarrow findFirst(t \in T \rightarrow t.id = b.id)$;
if $t = null$ **then**
  $t \leftarrow findFirst(t \in T \rightarrow t.body = b.body)$;
end
if $t = null$ **then**
  $t \leftarrow findFirst(t \in T \rightarrow t.body \approx b.body \wedge t.id \approx b.id)$;
end
if $t = null$ **then**
  $t \leftarrow findFirst(t \in T \rightarrow t.body = substring(b.body) \vee b.body = substring(t.body))$;
end
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;
```

2.1.3 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 = conflit(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
```

2.2 Initialization Blocks Handler

2.2.1 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_B | 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
```

2.3 Multiple Initialization Blocks Handler

2.3.1 Handler Algorithm

Algorithm 14: Handle

Input: L, B, R, M

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

$E_L \leftarrow$ `editedNodes(`IB_L, IB_B`)`;
$E_R \leftarrow$ `editedNodes(`IB_R, IB_B`)`;

$DEL_L \leftarrow$ `deletedNodes(`IB_L, IB_B, E_L`)`;
$DEL_R \leftarrow$ `deletedNodes(`IB_R, IB_B, E_R`)`;

foreach $b \in IB_B$ **do**
    $l \leftarrow E_L[b]$;
    $r \leftarrow E_R[b]$;
    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**
            $r \leftarrow find(r \in DEL_R \rightarrow r.body = b.body)$;
            if $r \neq null$ **then** `removeNode(`b, M`)` ;
        else
            $l \leftarrow find(l \in DEL_L \rightarrow l.body = b.body)$;
            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

$ADD_L \leftarrow$ `addedNodes(`IB_L, IB_B, E_L`)`;
$ADD_R \leftarrow$ `addedNodes(`IB_R, IB_B, E_R`)`;

$DEP \leftarrow$ `dependentNodes(`ADD_L, ADD_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 $l \in ADD_L$ **do**
    foreach $r \in ADD_R$ **do**
        if $l.body = r.body$ **then**
            $m \in find(m \in M \rightarrow m.body = r.body)$;
            `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, E
Output: set of initilization block nodes added by branch

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

Algorithm 17: Deleted Nodes

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

$D \leftarrow \{b \in IB_B \mid (\neg\exists n \in IB)(b.body = n.body)\}$;
$D \leftarrow \{n \in D \mid (\neg\exists e \in E)(n.body = e.key.body)\}$;
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: ADD_L, ADD_R
Output: map associating an added left node l in ADD_L and all added right nodes r in ADD_R with common global variables

$DEP \leftarrow \emptyset$;
foreach $l \in ADD_L$ **do**
  $DEP \leftarrow DEP \cup \{l : \emptyset\}$;
  $V_L \leftarrow$ `globalVariables(`l`)`;
  foreach $r \in ADD_R$ **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;

2.4 Type Ambiguity Error Handler

2.4.1 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  M_U ← textualMerge(treeToText(L), treeToText(B), treeToText(R));
5  I_L, I_R ← extractInsertions(M_U);
6  cs ← extractConflicts(M_U);
7  c ← compile(M_U);
8  ps ← problems(c);
9  foreach l ∈ ID_L do
10     m_l ← extractPackageMember(l.body);
11     foreach r ∈ ID_R do
12         m_r ← extractPackageMember(r.body);
13         if m_l = m_r then
14             p ← importDeclarationsProblem(l, r, ps);
15             if p ≠ null then
16                 m ← find(m ∈ M → m.body = l.body);
17                 m.body ← conflict(l.body, ε, r.body);
18                 m ← find(m ∈ M → m.body = r.body);
19                 removeNode(m, M);
20                 ps ← ps − p;
21                 break;
22             end
23         else if (m_l = ∗ ∨ m_r = ∗) ∧ importDeclarationsConflict(l, r, cs) then
24             I ← I_L;
25             m ← m_r;
26             if m_l ≠ ∗ then
27                 I ← I_R;
28                 m ← m_l;
29             end
30             i ← find(i ∈ I → import ∉ i ∧ m ∈ i);
31             if i ≠ null then
32                 m ← find(m ∈ M → m.body = l.body);
33                 m.body ← conflict(l.body, ε, r.body);
34                 m ← find(m ∈ M → m.body = r.body);
35                 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 ∈ 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 a 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;
```