1 Semistructured Merge

1.1 Early Concepts

1. Every node's origin is set to UNKNOWN beforehand

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);
$D_L \leftarrow \{b \in B \mid (\neg\exists l \in L)(l.id = b.id)\}$;
$D_R \leftarrow \{b \in B \mid (\neg\exists r \in R)(r.id = b.id)\}$;
$D_B \leftarrow D_L \cap D_R$;
foreach $d \in D_B$ **do**
  `removeNode`(d, M);
end
`updateLeafBodies`(M);
return M;

Algorithm 3: Update Leaf Bodies

Input: T

foreach $t \in T.children$ **do**
  `updateLeafBodies`(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

```
1  if A = null then return B ;
2  if B = null then return A ;
3  if A.type ≠ B.type ∨ A.id ≠ B.id then return null ;
4  M.id ← A.id;
5  M.type ← A.type;
6  M.origin ← B.origin;
7  M.children ← ∅;
8  if A.children = ∅ ∧ B.children = ∅ then
9  |  if MARKER ∈ A.body then
10 |  |  M.body ← A.body + B.body;
11 |  else if A.origin = LEFT ∧ B.origin = BASE then
12 |  |  M.body ← MARKER + A.body + SEPARATOR + B.body + SEPARATOR;
13 |  else if A.origin = LEFT then
14 |  |  M.body ← MARKER + A.body + SEPARATOR + SEPARATOR + B.body;
15 |  else
16 |  |  M.body ← MARKER + SEPARATOR + A.body + SEPARATOR + B.body;
17 |  end
18 |  return M;
19 else if A.children ≠ ∅ ∧ B.children ≠ ∅ then
20 |  foreach b ∈ B.children do
21 |  |  a ← find(a ∈ A.children → a.type = b.type ∧ a.id = b.id);
22 |  |  if a.origin = UNKNOWN then a.origin ← A.origin ;
23 |  |  if b.origin = UNKNOWN then b.origin ← B.origin ;
24 |  |  M.children ← M.children ∪ mergeNodes(a, b, step);
25 |  end
26 |  foreach a ∈ A.children do
27 |  |  b ← find(b ∈ B.children → a.type = b.type ∧ a.id = b.id);
28 |  |  if a.origin = UNKNOWN then a.origin ← A.origin ;
29 |  |  if b = null then M.children ← M.children ∪ a ;
30 |  end
31 |  return M;
32 end
33 return null;
```

2 Renaming Handler

2.1 Early Concepts

2.1.1 Possibly renamed without body changes nodes

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

2.1.2 Possibly deleted or renamed with body changes nodes

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

2.1.3 Nodes IDs similarity

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

2.2 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

if $l \neq null$ **then**
  return $find(m \in M \rightarrow m.id = l.id)$;
end
if $r \neq null$ **then**
  return $find(m \in M \rightarrow m.id = r.id)$;
end
return *null*;

2.3 Handle Algorithms

Algorithm 8: Check References and Merge Methods Variant

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

```
if l.id = b.id ∨ r.id = b.id then
    m.body = textualMerge(l, b, r);
    removeUnmatchedNode(l, r, m, M);
else if l.id ≠ r.id then
    m.body = conflit(l, b, r);
    removeUnmatchedNode(l, r, m, M);
else if l.body ≠ r.body then
    if newReferenceTo(l) ∨ newReferenceTo(r) then
        m.body = conflict(l, b, r);
    else
        m.body = textualMerge(l, b, r);
    end
    removeUnmatchedNode(l, r, m, M);
end
```

Algorithm 9: Merge Methods Variant

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

```
m.body = textualMerge(l, b, r);
removeUnmatchedNode(l, r, m, M);
```

Algorithm 10: Check Textual and Keep Both Methods Variant

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

```
if l.id = b.id ∨ r.id = b.id then
    if textualMergeHasConflictInvolvingSignature(b) then
        m.body = conflict(l, b, r);
        removeUnmatchedNode(l, r, m, M);
    end
else if l.id ≠ r.id ∧ l.body = r.body then
    m.body = conflict(l, b, r);
    removeUnmatchedNode(l, r, m, M);
end
```

Algorithm 11: Keep Both Methods Variant

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

```
if (l.id = b.id ∨ r.id = b.id) ∧ hasConflict(m) then
    removeConflict(m);
end
```

Algorithm 12: Remove Unmatched Node

Input: l, r, m, M

```
if l.id = m.id ∧ r.id ≠ m.id then
    removeNode(r, M);
end
```