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

Algorithm 3: Run Textual Merge On Leaves

Input: T

foreach $t \in T.children$ **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

```
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 Handlers

2.1 Early Concepts

1. **Nodes added by left**:
   $A_L \leftarrow \{l \in L \mid (\neg\exists b \in B)(l.id = b.id)\}$
2. **Nodes added by right**:
   $A_R \leftarrow \{r \in R \mid (\neg\exists b \in B)(r.id = b.id)\}$
3. **Nodes deleted from base**:
   $D_B \leftarrow \{b \in B \mid (\neg\exists l \in L)(b.id = l.id) \wedge (\neg\exists r \in R)(b.id = r.id)\}$

2.2 Renaming Handler

2.2.1 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 similariy**:
   $a.id \approx b.id \leftrightarrow a.id.name = b.id.name \vee a.id.params = b.id.params$

2.2.2 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 $null$;

2.2.3 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
```

2.3 Initialization Blocks Handler

2.3.1 Handler Algorithm

Algorithm 13: Handle

Input: L, B, R, M

$IB_L \leftarrow \{n \in A_L \mid n.type = INITBLOCK\}$;
$IB_R \leftarrow \{n \in A_R \mid n.type = INITBLOCK\}$;
$IB_B \leftarrow \{n \in D_B \mid n.type = INITBLOCK\}$;
$matches \leftarrow \emptyset$;
if $|IB_L| = 1 \wedge |IB_B| == 1 \wedge |IB_R| = 1$ **then**
  $matches \leftarrow matches \cup (IB_{L_1}, IB_{B_1}, IB_{R_1})$;
else
  foreach $b \in IB_B$ **do**
    $l \leftarrow findFirst(l \in IB_L \rightarrow l.body \approx b.body)$;
    $r \leftarrow findFirst(r \in IB_R \rightarrow r.body \approx b.body)$;
    $IB_L \leftarrow IB_L - l$;
    $IB_R \leftarrow IB_R - r$;
    if $l \neq null \wedge r \neq null$ **then**
      $matches \leftarrow matches \cup (l, b, r)$;
    end
  end
  foreach $l \in IB_L$ **do**
    $r \leftarrow findFirst(r \in IB_R \rightarrow r.body \approx l.body)$;
    $IB_R \leftarrow IB_R - r$;
    if $r \neq null$ **then**
      $matches \leftarrow matches \cup (l, null, r)$;
    end
  end
end
foreach $(l, b, r) \in matches$ **do**
  $m \leftarrow find(m \in M \rightarrow m.body = l.body)$;
  $m.body \leftarrow$ `textualMerge(`*l.body, b.body, r.body*`)`;
  $m \leftarrow find(m \in M \rightarrow m.body = r.body)$;
  `removeNode(`*m, M*`)`;
end

2.4 Multiple Initialization Blocks Handler

2.4.1 Handler Algorithm

Algorithm 14: 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  E_L ← editedNodes(IB_L, IB_B);
5  E_R ← editedNodes(IB_R, IB_B);
6  DEL_L ← deletedNodes(IB_L, IB_B, E_L);
7  DEL_R ← deletedNodes(IB_R, IB_B, E_R);
8  foreach b ∈ IB_B do
9      l ← E_L[b];
10     r ← E_R[b];
11     if l ≠ null ∧ r ≠ null then
12         updateMergeTree(l, b, r, M);
13     else if l ≠ null ∨ r ≠ null then
14         if l ≠ null then
15             r ← find(r ∈ DEL_R → r.body = b.body);
16             if r ≠ null then removeNode(b, M) ;
17         else
18             l ← find(l ∈ DEL_L → l.body = b.body);
19             if l ≠ null then removeNode(b, M) ;
20         end
21         updateMergeTree(l, b, r, M);
22     else
23         m ← find(m ∈ M → m.body = b.body);
24         removeNode(m, M);
25     end
26 end
27 ADD_L ← addedNodes(IB_L, IB_B, E_L);
28 ADD_R ← addedNodes(IB_R, IB_B, E_R);
29 DEP ← dependentNodes(ADD_L, ADD_R);
30 foreach (l, rs) ∈ DEP do
31     s ← ε;
32     foreach r ∈ rs do
33         s ← s + r.body;
34         m ← find(m ∈ M → m.body = r.body);
35         removeNode(r, M);
36     end
37     m ← find(m ∈ M → m.body = l.body);
38     m.body ← conflict(l.body, ε, s);
39 end
40 foreach l ∈ ADD_L do
41     foreach r ∈ ADD_R do
42         if l.body = r.body then
43             m ∈ find(m ∈ M → m.body = r.body);
44             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

$D \leftarrow \{d \in IB_B \mid (\neg\exists a \in IB)(d.body = a.body)\}$;
$A \leftarrow \{a \in IB \mid (\neg\exists d \in IB_B)(a.body = d.body)\}$;
$matches \leftarrow \emptyset$;
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;