A Category-Theoretic Approach to Syntactic Software Merging

Nan Niu, Steve Easterbrook, and Mehrdad Sabetzadeh

Department of Computer Science
University of Toronto, Canada
Email: {nn, sme, mehrdad}@cs.toronto.edu

September 27, 2005
Outline

- Introduction and motivation
  - Why do we need to merge software artifacts?
  - What’s wrong with traditional textual merging?

- Our approach to syntactic software merging
  - Mathematical underpinnings
  - Three steps by using our framework

- A proof-of-concept example

- Concluding remarks
Why merging?

- In large-scale software development & maintenance
  - Parallel changes
  - Optimistic version control mechanism

version 1

version 1a

version 1b

merged version

to yield a consistent shared view
Drawbacks of textual merging

- Only very basic conflicts can be identified due to the lack of structured syntactic and semantic knowledge

- Only at the source code level

- Now, let’s look at a three-way merging example
### version 1

1 ... void enter_game_record (char *msg) {
   ...
   /* AWARD_P is a constant */
5   if (get_point (msg) > AWARD_P)
      send_award (msg);
   ...
}
10 int get_point (char *msg) {
    /* return game point */
    ...
}
15 void send_award (char *msg) {
    /* send plain text email */
    ...
}

### version 1a

... void enter_game_record (char *msg) {
   ...
   /* award policy is modified */
5   if (pass_rank (msg))
      send_award (msg);
   ...
}
...
10 int pass_rank (char *msg) {
    /* a boolean function */
    /* return 0 or 1 */
    ...
}
...

### version 1b

... void send_award (char *msg) {
    /* send plain text email */
    ...
    /* check game point */
    int p = get_point (msg);
    /* send HTML email */
    ...
}
...

### merge of versions 1a and 1b

1 ... void enter_game_record (char *msg) {
   ...
   /* award policy is modified */
5   if (pass_rank (msg))
      send_award (msg);
   ...
}
10 int pass_rank (char *msg) {
    /* a boolean function, return 0 or 1 */
    ...
}
15 int get_point (char *msg) {
    /* return game point */
    ...
}
20 void send_award (char *msg) {
    /* send plain text email */
    /* check game point */
    int p = get_point (msg);
    /* send HTML email */
    ...
}
Highlights of our approach

- Only very basic conflicts can be identified due to the lack of structured syntactic and semantic knowledge
  - Let’s structure this evolutionary knowledge
- Only at the source code level
  - Let’s use graph-based representations as an aid to comprehension
- Also want to support software reverse engineering and reengineering
Graphs and graph homomorphisms

version 1a

$G_A$
- pass_rank
- enter_game_record
- send_award

version 1b

$G_B$
- get_point
- send_award

version 1

$G_C$
- enter_game_record
- get_point
- send_award

merged result

$G_B$
- pass_rank
- enter_game_record
- get_point
- send_award

$G_P$
- pass_rank
- enter_game_record
- get_point
- send_award
A partial order is a reflexive, antisymmetric, and transitive binary relation. A non-empty set with a partial order on it is called a poset.

In a poset, if the least upper bound (supremum) and the maximal lower bound (infimum) exist for any pair of elements, then the poset is called a lattice.

If the supremum and the infimum exist for any subset of the poset, then the poset is a complete lattice.
Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings

Evolutionary orderings
A pushout of two morphisms $f: C \to A$ and $g: C \to B$ in a category is the combination of $A$ and $B$ with respect to a shared part $C$.

Pushout formalizes the three-way merging in software evolution.
Pushout in the running example
Fuzzy sets

⇒ Want to label nodes and arrows in graphs with certain degrees
  ⇧ Values drawn from a partial order to express the evolutionary ordering

⇒ Appeal to fuzzy set theory
  ⇧ For a poset A, an **A-valued set** is a pair \((S, \sigma)\) where:
    ⇒ \(S\) is a set called the **carrier set** of \((S, \sigma)\)
    ⇒ \(\sigma: S \to A\) is a function assigning a **degree of membership** from \(A\) to every \(s\) in \(S\)
  ⇧ A morphism \(f: (S, \sigma) \to (T, \tau)\) is a function \(f: S \to T\) s.t. \(\sigma \leq \tau \circ f\)
    ⇒ i.e. the degree of membership of \(s\) in \((S, \sigma)\) is **NOT** larger than that of \(f\) in \((T, \tau)\)
  ⇧ The above definitions give rise to a category denoted by \(\text{Fuzz}(A)\)
Example of fuzzy graphs
Results of fuzzy set categories

- **Theorem:** All the pushouts exist for Fuzz(A) when A is a complete lattice

- **Procedure for computing the pushout of a pair of Fuzz(A)-morphisms:**
  - Compute the pushout of the carrier morphisms
  - Compute a membership degree for every node and arrow in the pushout graph by taking the supremum of the membership degrees of all those elements that are mapped to this particular node and arrow
Three steps in our approach

- **Preprocessing**
  - Transform source codes into graph-based software artifacts

- **Pushout diagram construction**
  - Most important step
  - Auxiliary tools are used

- **Pushout computation and analysis**
  - Can be totally automated

☆ As a semi-automated comprehension task
Preprocessing

source code $\xrightarrow{Rigi}$ graph $\xleftarrow{projection}$ subgraph

system $\xleftrightarrow{inclusion}$ system $\xleftrightarrow{inclusion}$ subsystem
Pushout construction in $Fuzz(A_{13})$

(a) $G_A = (G'_A \cup G_1)$, $\phi \uparrow$

(b) $G'_A \xrightarrow{\psi'} G_P$ (merged result)

(c) $G'_B \xrightarrow{\psi} G_B = (G'_B \cup G_1)$
Assigning membership degrees

⇒ Label every element in $G_1$ of $G_C$ with $\text{MM}$;

⇒ Label every element in $G_{\text{new}}$ with $\text{AA}$;

⇒ Label every element in $G_A \setminus G'_A$ (resp. $G_B \setminus G'_B$) with $\text{RM}$ (resp. $\text{MR}$);

⇒ Label every element in the image of the carrier function $\Phi(G_{\text{new}})$ (resp. $\Psi(G_{\text{new}})$) with $\text{AA}$;

⇒ Label every element in $G_A \setminus G_C$ (resp. $G_B \setminus G_C$) with $\text{AM}$ (resp. $\text{MA}$); and

⇒ Label all remaining elements in $G_A$ (resp. $G_B$) with $\text{PM}$ (resp. $\text{MP}$).

☆ Element means node and arrow of a graph component-wise
Pushout computation and analysis

A system of interconnected graphs in Fuzz($A_{13}$) is **syntactically inconsistent** if the merged result has some node or arrow with an inconsistent truth value.

It’s up to the maintainers to designate (in)consistent values from $A_{13}$, e.g.

- **total agreement**: RR, PP, AA as consistent
- **explicit conflicts**: PR, RP, T as inconsistent
- **three-way consolidation merging**: PR, RP, T, RR, AA as inconsistent
A proof-of-concept example

- A commercial proprietary system (iBBS) developed in-house by a single company in Beijing, PR China

- iBBS’s development began in the early 1990s
  - Written in C
  - Microsoft SourceSafe as version control
Call graphs of iBBS (version 1b)

119 functions (nodes), 75 calls (arcs)

(a) overall system

28 functions (nodes), 43 calls (arcs)

(b) “util” subsystem
## Comparison of “util” programs

<table>
<thead>
<tr>
<th>version</th>
<th>1</th>
<th>1a</th>
<th>1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>overview</td>
<td>WIN32</td>
<td>fix bugs</td>
<td>Linux86</td>
</tr>
<tr>
<td>LOC (lines of code)</td>
<td>2,159</td>
<td>2,476</td>
<td>2,722</td>
</tr>
<tr>
<td>number of functions</td>
<td>13</td>
<td>19</td>
<td>28</td>
</tr>
<tr>
<td>number of calls</td>
<td>10</td>
<td>27</td>
<td>43</td>
</tr>
<tr>
<td>cyclomatic complexity</td>
<td>7</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>
Interconnecting the call relations
Pushout – the merged result
Concluding remarks

⇒ Summary

⇒ A formal framework for three-way syntactic software merging
  ➢ Graph-based notations
  ➢ Structured evolutionary properties
  ➢ An exploratory lattice – $A_{13}$

⇒ Limitations and weaknesses
  ➢ Based purely on syntactic mappings
  ➢ Hard to build interconnections between software artifacts

⇒ Future work

⇒ Support semantic & structure software merging
⇒ Add typing constraints to handle heterogeneous nodes and arrows in one view
⇒ Automate the identification of interconnections
Acknowledgments

👍 Partner company, esp. Hongying Zhao

👋 Kenny Wong, Li-Shih Huang & anonymous ICSM’05 referees

💰 Financial support was provided by NSERC & MITACS

THANK YOU!