Recovering the Evolution Stable Part using an ECGM algorithm: is there a Tunnel in Mozilla?

Sègla Kpodjedo*, Filippo Ricca**, Philippe Galinier* and Giuliano (Giulio) Antoniol*

*SOCCER Lab. – Ecole Polytechnique de Montreal, Quebec, Canada
** CINI Unity at DISI, University of Genoa, Italy
The context

- The design evolution is an important “knowledge base”
  - Programmers: find solutions of similar problems and how they have been resolved in the past
  - Managers: find evolution patterns symptomatic of problems in order to avoid them in the future

- Finding changes in long-lived applications is a non trivial task. It depends mainly on
  - the modeled software artifacts and their representations
  - the size of the system
  - the length of its history

- Several proposals based on Graph and Tree matching
  - Focused on: class diagram evolution
  - Difficult to adapt to a different class of problems
    - e.g., evolution of state or interaction diagrams
Goal: investigating the applicability of ECGM algorithms for studying software evolution
   - Class diagram evolution

How: using Mozilla as case study

Steps:
1. Mozilla snapshots were collected every 15 days over the last six years
   - 144 subsequent snapshots
2. Recover class diagrams - reverse engineering
3. Model class diagrams as graphs
4. Apply ECGM to subsequent graphs
5. Evolution information:
   - matched classes
   - matched relations
   - insertions and removals of classes/relations
Error Correcting Graph Matching

- An ECGM algorithm:
  - allows matching two nodes that violate constraints such as:
    - node labels
    - exact correspondence of edges
    - edge labels

- NP-hard problem – exact algorithms prohibitive computation times.
  - Resorting to meta-heuristic search strategies
    - a penalty is assigned to each violations
    - ECGM finds the “best” matching
      - minimizes the overall penalty cost (cost function)
Pros and cons

**Pros:**
- ECGM an elegant framework - no assumption on problem structure.
  - It works with graphs!
- ECGM easily tailored to different classes of problems
  - fitting ECGM cost weights.
    - Our case: studying class diagram evolution

**Cons:**
- ECGM are meta-heuristic search strategies and thus:
  - the optimum cannot be guaranteed
  - different runs may produce slightly different results
An ECGM indicate edit operations to transform the first graph into the second

- Matching $MG_1$ to $MG_2$ => « errors »
- Deletion of $D_{G_1}$: the nodes and all their adjacent edges
- Insertion of $I_{G_2}$: the nodes and all their adjacent edges
Modeled Errors

- Node matching errors ($M_{G1}$ to $M_{G2}$)
  - Dissimilarity of related information (labels)

![Diagram of node matching errors](xyz \rightarrow yxw)

- Edge matching errors ($M_{G1}$ to $M_{G2}$)
  - Structural error: an edge present in only one graph

![Diagram of edge matching errors](G1 \rightarrow G2 and G1 \leftarrow G2 or G1 \rightarrow G2 and G1 \leftarrow G2)

- Label error: different edges from one graph to the other

![Diagram of label error](G1 \rightarrow a \rightarrow G2 and G1 \leftarrow b \leftarrow G2)
Our ECGM algorithm

- Our ECGM implementation relies on a Tabu Search (TS) guided by:
  - **local information**: node features - number of incoming/outgoing edges.
  - **global information**: provided by the PageRank algorithm.
- PageRank measures the relative importance of each element of a hyperlinked set
  - it assigns a numerical weighting to nodes
  - the more references a vertex gets from other elements, the more importance it deserves.

The nodes 6, 1, 10, 8 appear to be the most important
Fitness function parameters

- **Fitness** based on $c_{nm}$, $c_{no}$, $c_{eo}$, $c_{ems}$, $c_{eml}$:
  - node matching error ($c_{nm}$)
  - node deletion and insertion ($c_{no}$)
  - edge deletion/insertion applied to edges of deleted/added nodes ($c_{eo}$)
  - edge matching:
    - edge structural error - an edge is inserted/deleted between two matched nodes ($c_{ems}$)
    - edge label error ($c_{eml}$)
      - for example, an association is mapped into an aggregation

- **Fitness**: the sum of the cost of edit operations to transform the first graph into the second.
Class diagram evolution as ECGM

- **Nodes**
  - classes
    - Class name, number of fields and methods

- **Edges**
  - Relations: Inheritance, Association, Aggregation

To use our ECGM algorithm we need a way to measure the dissimilarity between two nodes.

- C1(l1,#m1,#a1) and C2(l2,#m2,#a2)

\[
\text{Dissim}(C1, C2) = \left[ l_w \left( \frac{\text{Sim}(l_1, l_2)}{\max(|l_1|, |l_2|)} \right) + m_w \left( \frac{|#m_1 - #m_2|}{\max(#m_1, #m_2)} \right) + a_w \left( \frac{|#a_1 - #a_2|}{\max(#a_1, #a_2)} \right) \right] \]

where \( l_w, m_w, a_w \) are respectively weights for labels, methods and attributes.

- \( \text{node\_matching\_cost}(C1, C2) = c_{nm} \times \text{Dissim}(C1, C2) \)

"Sim(l1, l2) is the Levenshtein distance between the two strings l1 and l2"
Tuning ECGM weights

- Six versions of the LaTazza application
  - a beverage vending machine
  - 17 Java classes and 37 relations
  - about 6184 LOC

**Assumption:** the tuned parameters deduced from LaTazza can be reused “as is”.

Iterative procedure (trial and error):
1. determine/tune \((c_{nm}, c_{no}, c_{eo}, c_{ems}, c_{eml}, l_w, m_w, a_w)\);
2. put \(I=1\);
3. If \(I=6\), stop; otherwise, run our ECGM algorithm on the two subsequent versions \((V_I, V_{I+1})\) of LaTazza;
4. check the result; if it is satisfactory, put \(I=I+1\) and Goto step 3; otherwise Goto step 1;

Finally:
- \((c_{nm}, c_{no}, c_{eo}, c_{ems}, c_{eml}, l_w, m_w, a_w) = (50, 25, 10, 30, 25, 70, 25, 5)\)
The case study: Mozilla

- Open source suite implementing a Web browser plus other tools.
- Mostly in C++ - C only a small fraction
- All components in the Mozilla suite have been extracted:
  - all configuration parameter of the CVS Mozilla client that retrieves CVS snapshot
    - SeaMonkey, mail/news, composer and other applications
- Snapshots collected twice per month from Jan. 2002.
- Dec 2007:
  - 9,000 classes with 23,000 relations
  - Size 3.3MLOC about 11,950 files
Research questions

- **General:**
  - Is it possible to apply an ECGM algorithm to study the class diagram evolution of a large software application?
  - Can results of an ECGM algorithm provide useful insight to developers?

- **Specific:**
  - **RQ1 – ECGM Applicability:** is it possible to apply our ECGM algorithm to study the class diagram evolution of Mozilla?
    - more than 8,000 classes and 20,000 relations
  - **RQ2 – ECGM Usefulness:** is it possible to use our ECGM algorithm to find the Mozilla’s tunnel?
    - A class always matched in all the considered snapshots is considered to be stable and part of a *tunnel*.
  - **RQ3 – ECGM Stability:** are different runs of our ECGM algorithm producing consistent and stable results?

“Classes likely constituting the application’s backbone”

![Diagram of class diagram evolution with snapshots](image-url)
RQ1 – ECGM Applicability

- We can answer to this question in affirmative way.
- We applied our algorithm to 144 class diagrams of Mozilla
  - the computation time was acceptable a little more than 15 hours.
- The ECGM algorithm coded in C++ was compiled with g++ and run on a Linux Bi Processor Opteron 64-bit with 16 Gb RAM running RedHat Advanced Server version 4.
  - The average time per matching about 382 seconds
RQ2 – ECGM Usefulness (1)

- The answer to this question is yes.
- Our ECGM algorithm identified the Mozilla tunnel:
  - **Classes**: 5514 Mozilla classes out of 8295 of the first snapshot have small or no distortion.
    - 66% of the classes belong to the tunnel.
    - There are 2263 threads in the tunnel with zero cost
      - about 41%
  - **Relations**: 11489 out of 20289 edges present in the first Mozilla snapshot are in the tunnel
    - About 57% of the relations belong to the tunnel
    - 9540 kept the same value throughout it
      - The type of the relation is not changed
RQ2 – ECGM Usefulness (2)

- A thread starts at 1/1/2002 and stops when the algorithm does not find a class correspondence with the next snapshot.
- The bar chart representing the number of threads having length “X”
  - the tunnel is not represented
- Two points in time where several classes have a disrupt in structural relations
  - April 1, 2004 and Feb 1 2007
- The two peaks correspond to 486 threads of length 54 and 442 threads of length 122.
- What is happened in Mozilla?
CVS of Mozilla inspection:
- Actually 486 and 442 classes were removed between March 15 - April 1, 2004 and Jan 15 - Feb 1, 2007.

Why of such cleanup – future work.
- For example, between March 15 and April 1 in 2004 were removed from CVS repository
  - an address book implementation (libaddr),
  - an HTML editor (liblayout)
  - a message system (libmsg)
- These components and the contained classes were never part of Mozilla releases since the first Mozilla 1.0 release. **Why?**

No real documentation … we are currently investigating the reason why components not included in any release survived for years in CVS …
The answer to the third research question is yes.

Ten different runs of the ECGM algorithm on two different couples of class diagrams:
- closed by (Jan 01, 2002 and Jan 15, 2002)
- outermost (Jan 01, 2002 and Dec 15, 2007)

Stability measured via the Jaccard index:
- considering the matches in different runs augmented with the matching cost.

\[ J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \]

<table>
<thead>
<tr>
<th>common matches</th>
<th>closed by (01/01/2002-15/1/2002)</th>
<th>outermost (01/01/2002-15/12/2007)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>edges</td>
</tr>
<tr>
<td>matched</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>added</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>deleted</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Soundness and accuracy of results

- The complete manual verification is unfeasible.
- We considered:
  - *closed by* (Jan 01, 2002 and Jan 15, 2002)
  - *outermost* (Jan 01, 2002 and Dec 15, 2007)
- We resort on the following strategy:
  - automatically verified that whenever a matching between two classes has zero cost, the name of classes is the same, the number of attributes and methods as well as relations are unchanged.
    - No error was discovered for zero cost matches.
  - then, manually inspect a sample of classes and edges with non zero matching cost.
    - Inspected non zero cost matches were also correct.
Threats to validity

- **External validity** - generalization of our findings.
  - The scenario presented is realistic and representative of many real-world situations.
  - We believe that with appropriate tuning of the cost parameters, ECGM could be applied to other software artefacts such as call graphs, UML activity diagrams, ...

- **Construct validity**
  - E.g., due to errors introduced by measurement instruments.
    - Our strategy of reverse engineering class diagrams may contain imprecision
      - there is need to compare obtained results with other reverse engineering tools.
    - Another critical element is the choice of the cost weights
      - different costs produce different ECGM results ...
      - we plan to better assess the influence of cost weights in future work.

- **Reliability validity** - replicating the study.
  - Mozilla CVS is publicly available
  - details of the algorithm are presented in [10]
  - all data used can be downloaded from the SER repository.
Conclusions

- Class diagram evolution was modeled as an ECGM problem.

- By means of a large case study, we provided evidence of:
  - ECGM applicability
    - In about 15 hours we obtained the matching of 143 Mozilla’s class diagram pairs, each one containing about 10,000 classes and 20,000 relations.
  - ECGM usefulness
    - We automatically identified the Mozilla backbone classes
    - We also found two cleanup and restructuring points were classes never part of Mozilla releases but present in the CVS repository were removed.
  - ECGM stability
    - less than 1% error when class diagrams are taken from two snapshots six years apart.
Future works

- Considering the signature in its entirety and repeat the Mozilla’s experiment to highlight potential accuracy improvements.
- Validating the Mozilla case study:
  - computing manually the number of correctly identified changes and the stable classes produced by our algorithm during Mozilla’s evolution.
- Comparing our ECGM algorithm with the maximum match algorithm proposed in [1] and with UMLDiff [17].


Thanks for the attention!

Questions?
Tabu Search (TS) algorithm

- **Local search techniques.** Given a cost function \( f \) to be minimized over a set \( S \) (*the search space*), a local search technique starts from some initial point (*solution*) in the search space and proceeds iteratively (*moves*) from one point in \( S \) to another (*a neighbour*).

- There is no guarantee of obtaining an optimal solution as the search may get trapped in local optima.

- Ts, a meta-heuristic algorithm, tries to avoid local optima to find better solutions.
  - pejorative moves are permitted
  - some historical information related to the evolution of the search is kept (*tabu list*). Such an information will be used to guide the movement from one solution to the next one avoiding cycling.
  - other lists may be used to intensify the search in a promising area of the search space.
More information about our ECGM algorithm

- In a local search the choice of a move is essentially guided by the cost function
  - We want minimize it ....
  - However in our problem the number of possible choices is very large: it may lead to poor performance or low accuracy of the result.
    - tested experimentally
  - To make the search efficient we use two additional information (combined) to do the move:
    - Local information: node features such as the number of incoming/outgoing edges
    - Global information: provided by the PageRank algorithm
PageRank

- PageRank is used by our algorithm to guide the moves.
- PageRank basically measures the relative importance of each element of a hyperlinked set and assigns it a numerical weighting.
- Let:
  - $Nu = |Fu|$ be the number of links $u$ points to
  - $Bu$ be the set of nodes that point to $u$
  - $k$ be a factor used for normalization — to force the range of $PageRank(u)$ in $[0, 1]$.

$$PageRank(u) = k \times \sum_{v \in Bu} \frac{PageRank(v)}{Nv}$$

- Algorithm able to compute PageRank is based on random walks
Extracting class relations from code might have some degree of imprecision.

We extracted the class diagrams of Mozilla using our **home-built reverse engineering tool** via an intermediate representation.

- AOL [1]

The class diagram is extracted from C++, IDL, Java and header files.

- Header files are important since classes as well as C structures are almost always defined in header files and sometimes, in Mozilla, a C structure is used as root of a class hierarchy.

We reuse the same infrastructure of [1] and thus:

- an aggregation is recognized from code if and only if a template, an object array, or an instance of an object is declared as data member.
- All the remaining cases, object pointers and references both as data members and formal parameters to methods, generate associations.