Software Integration

John Businge

john.businge@unlv.edu

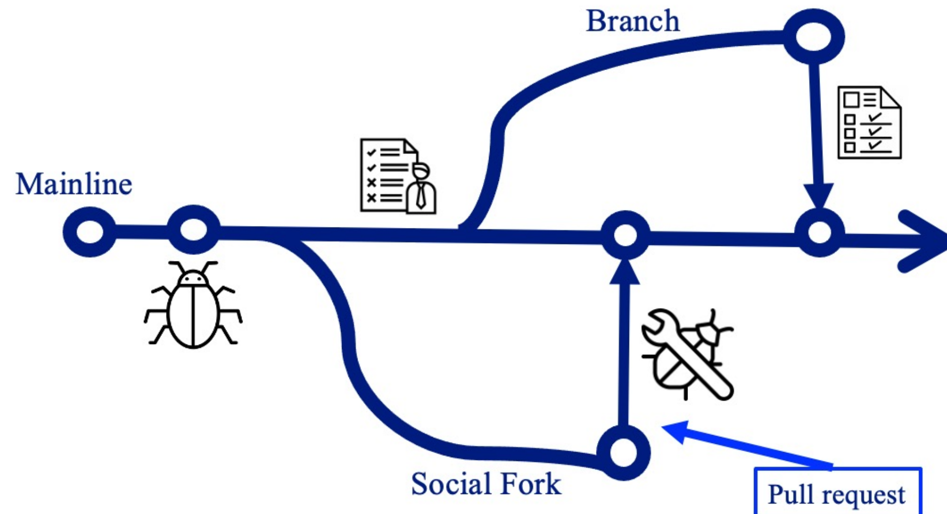
Version Control Systems

- Keep track of the software development history
- Became popular with the rise of distributed software development
- Offer practices that facilitate collaborative software development

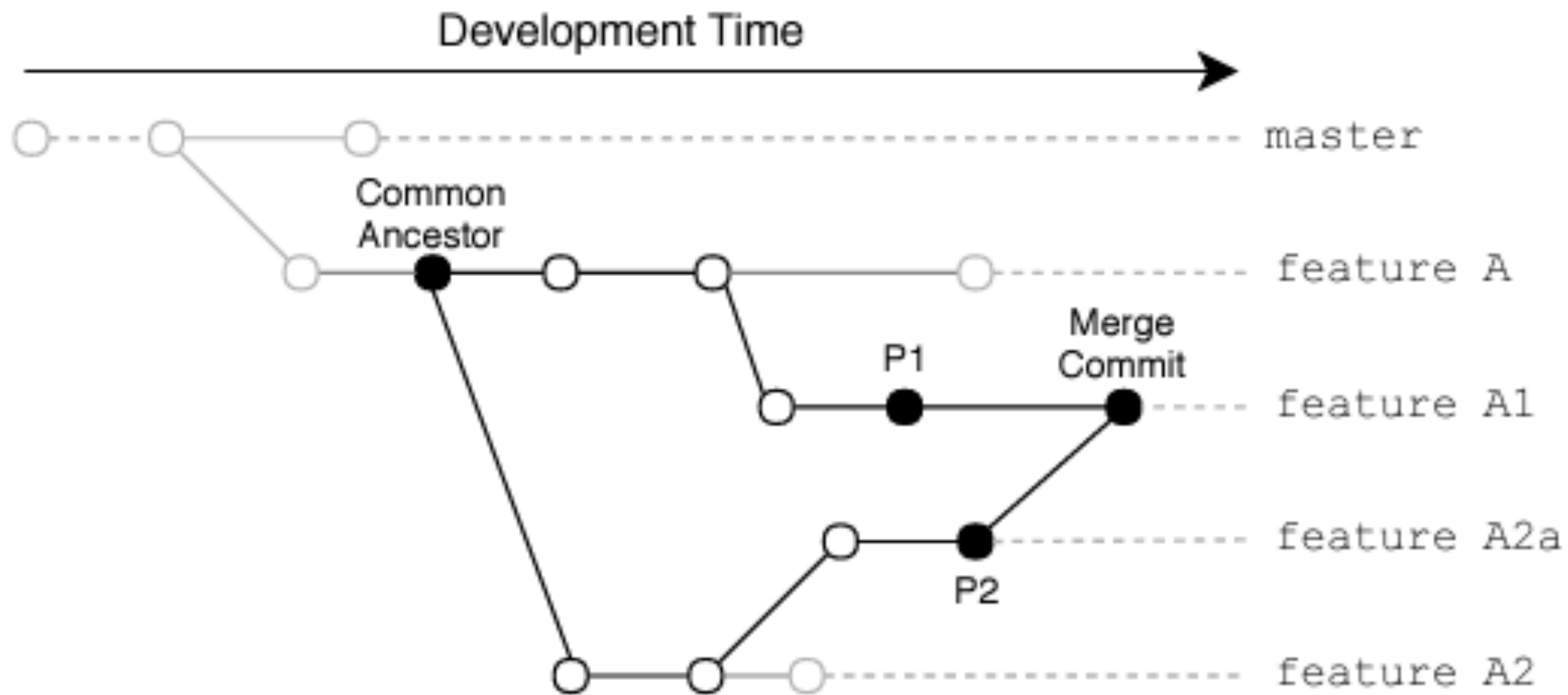
- Offer practices that facilitate collaborative software development

Branching/Forking

- A branch is an instance of the source code
- Developers create multiple branches and apply their changes in parallel
- Reasons for branching: isolating development work, bug fixes, releases, etc.



Merging/integation



Merge Scenario

Mahmoudi et al. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts, SANER 2019

Merge Conflict

- Merge conflicts may arise because of inconsistent changes to the code
- 16% of merge scenarios lead to conflicts [1]
- Developers have to resolve such conflicts before proceeding
- Wastes their time and distracts them from their main tasks
- Based on the nature of the merge scenario, a textual three-way merge tool, such as the one used by GIT might not be able to merge the two versions of a file automatically.

1. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts”

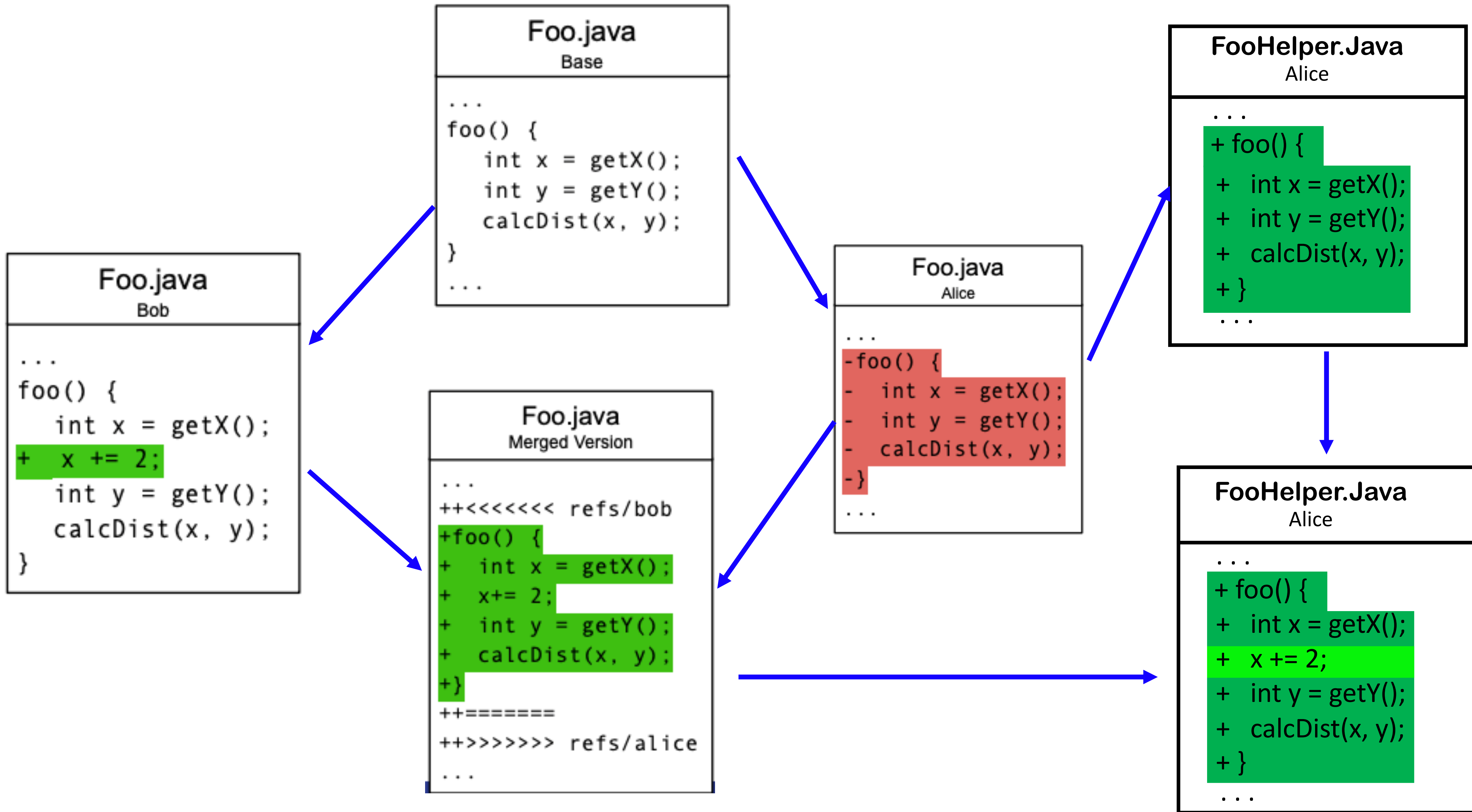
Merge Conflict

Content Level merge conflict

For a given conflicting merge scenario, GIT can report conflicts across multiple files. GIT categorizes conflicts into six types:

- **add/add**: When both merge, parents add a new file with the same name but with different contents.
- **content**: When both parents apply different changes to the same file in the same location.
- **modify/delete**: When *P1* modifies a file while *P2* deletes it.
- **rename/add**: When *P1* renames a file, and *P2* adds a new file with the same name.
- **rename/delete**: When *P1* renames a file, and *P2* deletes it.
- **rename/rename**: When both parents rename a file to different names.

File Level merge conflict



Foo.java
Base
...
foo() {
int x = getX();
int y = getY();
calcDist(x, y);
}
...
Foo.java
Bob
...
foo() {
int x = getX();
+ x += 2;
int y = getY();
calcDist(x, y);
}
Foo.java
Alice
...
-foo() {
- int x = getX();
- int y = getY();
- calcDist(x, y);
-}
...
FooHelper.Java
Alice
...
+ foo() {
+ int x = getX();
+ int y = getY();
+ calcDist(x, y);
+ }
...
Foo.java
Merged Version
...
++<<<<<<< refs/bob
+foo() {
+ int x = getX();
+ x+= 2;
+ int y = getY();
+ calcDist(x, y);
+}
++=======
++>>>>>>> refs/alice
...
FooHelper.Java
Alice
...
+ foo() {
+ int x = getX();
+ x += 2;
+ int y = getY();
+ calcDist(x, y);
+ }
...

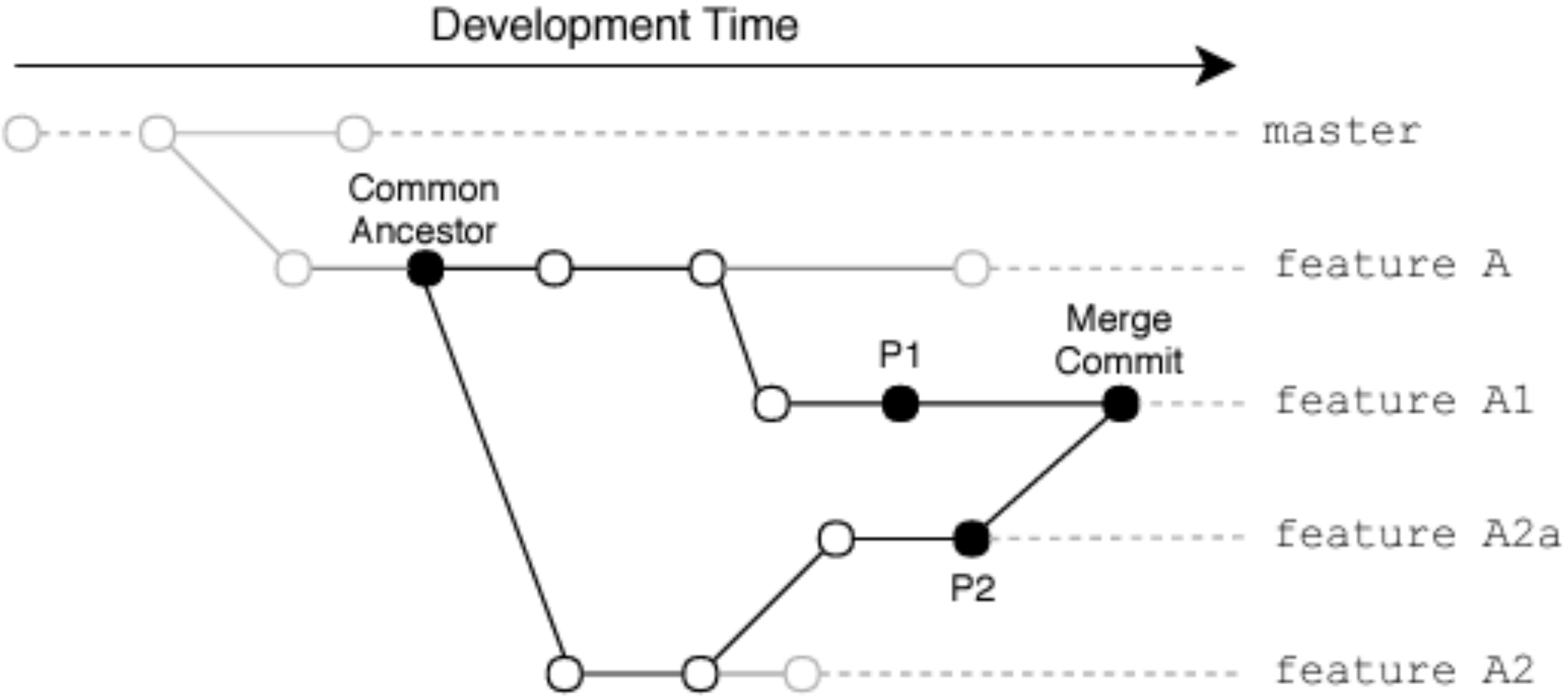
Refactoring Aware Operation-Based Merging

1. Detect conflicting regions
2. Detect evolutionary changes
3. Detect Refactorings
4. Detect involved refactorings in the conflicting region

https://github.com/ualberta-smr/RefactoringsInMergeCommits

Mahmoudi et al. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts (2019). [https://ieeexplore.ieee.org/document/8668012]

Merge Scenario



Step 1. Detecting Conflicting Regions

- First detect all merge scenarios
- Replay the merge scenario using the following commands:
  - git checkout P1
  - git merge P2
- If conflicting, git merge will report a list of conflicting files and their conflict types. The information is saved in the database.
- git diff command will report all conflicting regions for Java files with content conflicting types.
- All the information is recorded in the database

Step 1. Detecting Conflicting Regions

Find merge commits

Find conflicting regions

P1 P2 Conflicting region

```
@@@-65,1 -73,2 +78,5@@@
++<<<<<<< HEAD
 +int a = 2;
++=======
+ float f = .2;
+ int a;
++>>>>>>> refs/branch
```

- git merge reports
  - conflicting files
  - conflicting types
- git diff – conflicting region

Step 2. Detect evolutionary changes

- Tracking the historical evolution of the conflicting region between common ancestor (CA) and P1/P2.
- For each conflicting region, we detect all commits that have touched that region
- These commits are called evolutionary commits
- Use the following commands:

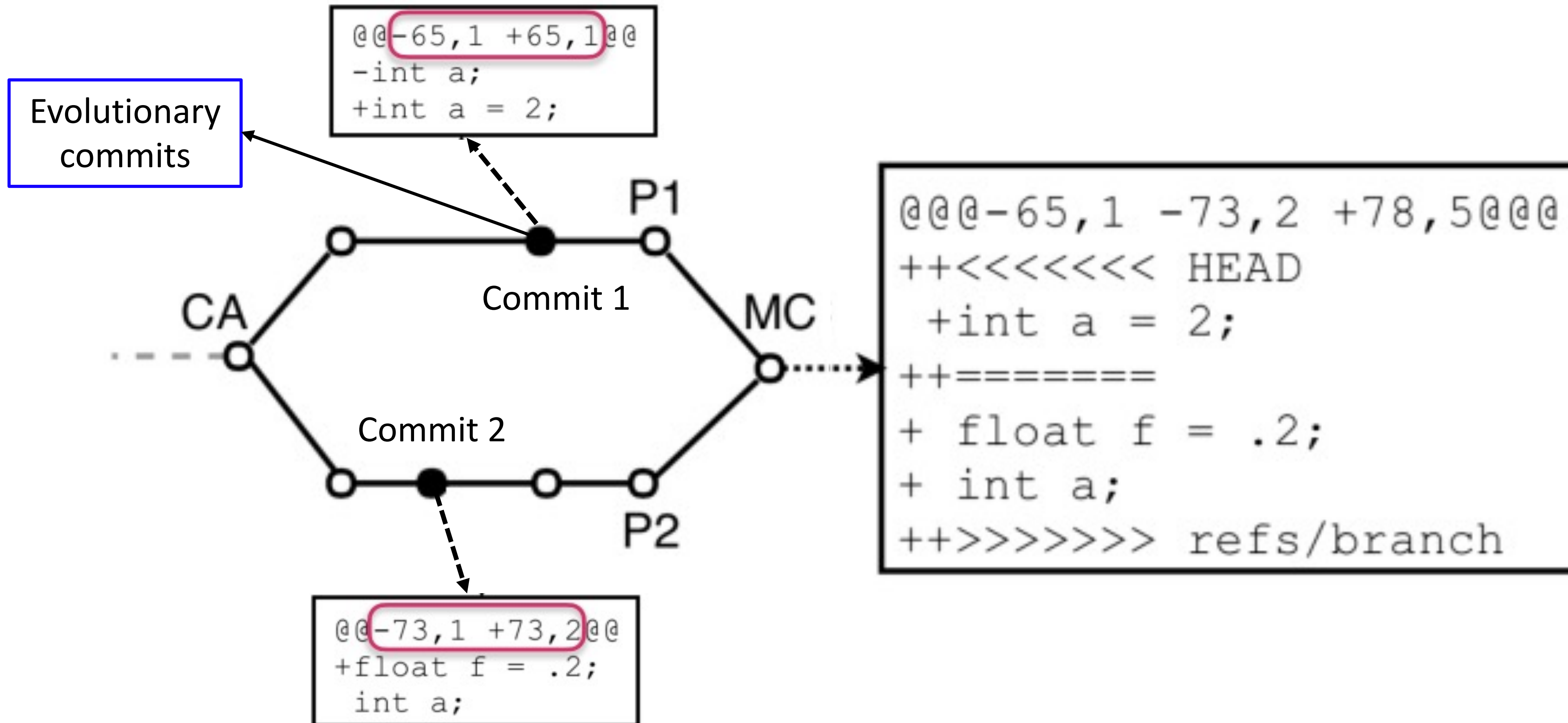
Revision range of file in P1

  - git log -L startP1,endP1:file P2..P1

includes all commits that are reachable from *P1* and not reachable from *P2*

  - git log -L startP2,endP2:file P1..P2

Step 2. Detect evolutionary changes



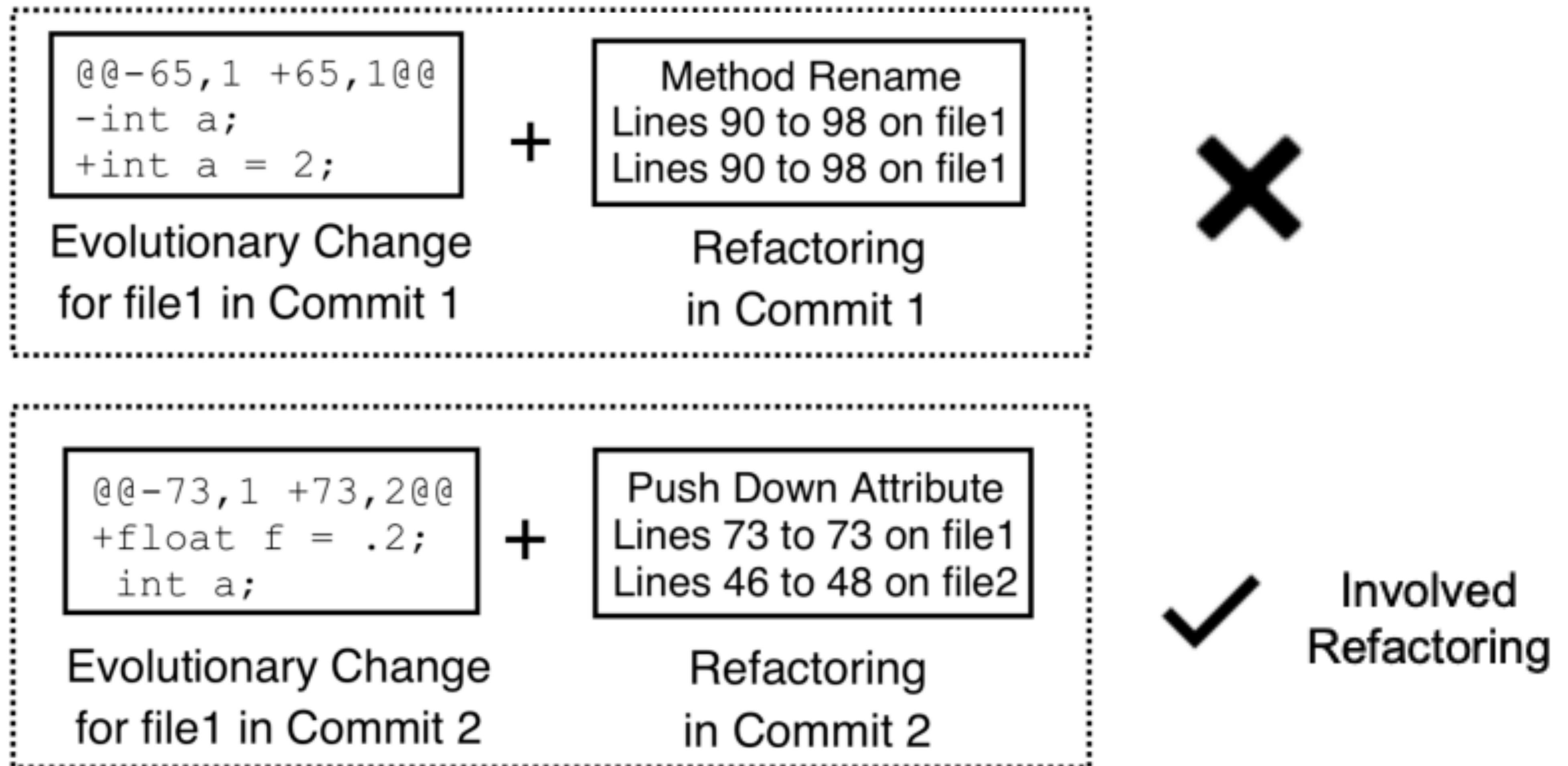
Step 3. Detect Refactorings

- RefactoringMiner is used in all the evolutionary commits
- We record the types and code regions of each refactoring to the database
- RefactoringMiner reports the files and the exact code ranges (with line numbers) that were touched by a refactoring operation.
- At least two code ranges for a refactoring change are stored: one code range corresponds to the refactored code element before refactoring, and the other corresponds to the element after refactoring.

Step 4. Detect involved refactorings

- The code range information for refactoring operations and evolutionary changes is used
- If there is an intersection between a refactoring and a refactoring, we call that refactoring an involved refactoring.

Step 4. Detect involved refactorings



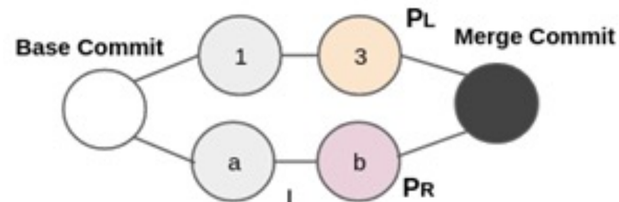
Step 4. Detect involved refactorings

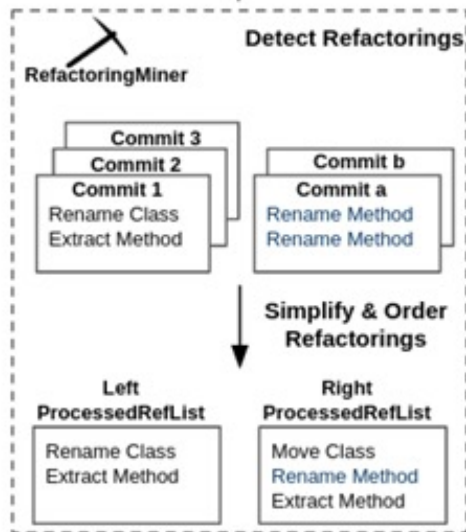
- In the final step, the identification of the refactoring operations that have affected the evolution of conflicting regions. In other words, we are trying to determine if an evolutionary change that later leads to a conflict contains a refactoring operation.
- Using the code range information for both refactoring operations (Step 3) and evolutionary changes to conflicting regions (Step 2), they determine if there is an overlap between them.
- They consider a refactoring and evolutionary change as *overlapping* if they have at least one line in common, either in their old-commit code ranges or in their new-commit code ranges.
- They call such refactoring operations that have overlapping code ranges with an evolutionary change *involved in refactoring operations* since they are involved in the changes that are related to the conflicting region.
- In the example of Figure, Step 4 shows that the refactoring in commit #1 would *not* be considered as an involved refactoring, while the refactoring in commit #2 would be considered so.

Refactoring-Aware tools

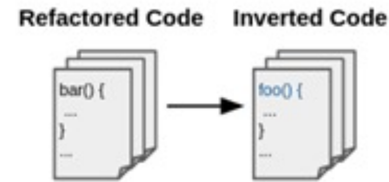
RefMerge

Step 1: Detect and Simplify Refactorings

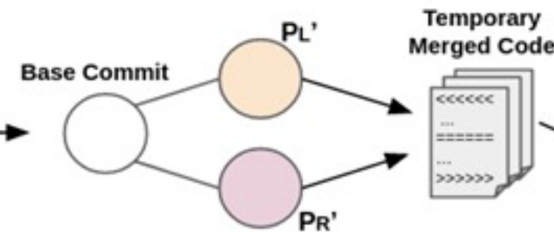




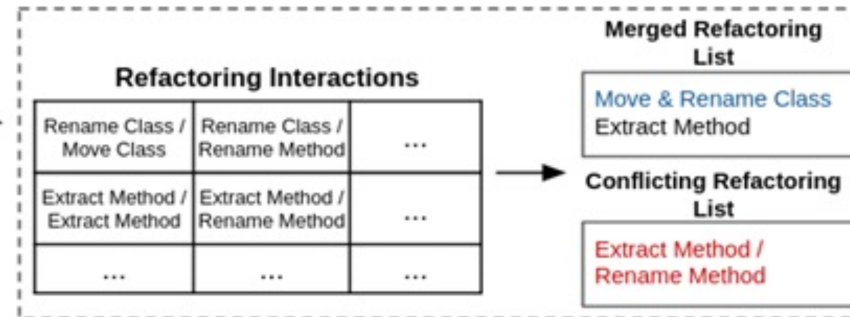
Step 2: Invert Refactorings



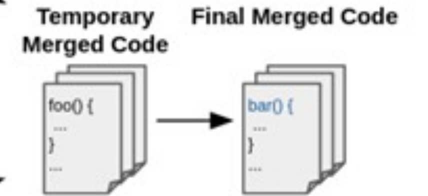
Step 3: Merge



Step 4: Detect Refactoring Conflicts



Step 5: Replay Refactorings



Ellis et at. A Systematic Comparison of Two Refactoring-aware Merging Techniques. 2022

https://github.com/ualberta-smr/RefactoringAwareMergingEvaluation

The Project

- You will use **RefactoringsInMergeCommits** with the **git cherry-pick** instead of **git merge**.
- You will employ the same approach of mining git logs to extract interesting data for the project.
- Let us go to the Lab.