Software Integration

John Businge

john.businge@unlv.edu

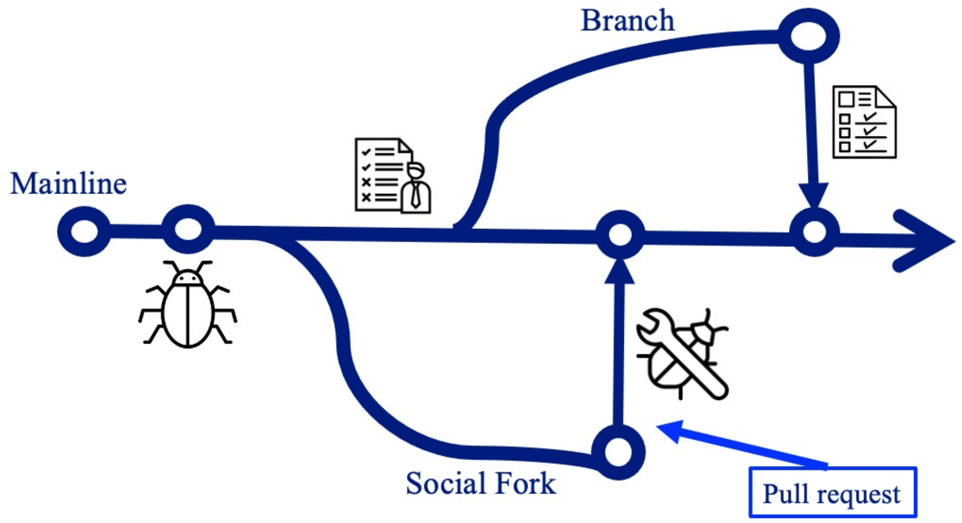
Version Control Systems

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

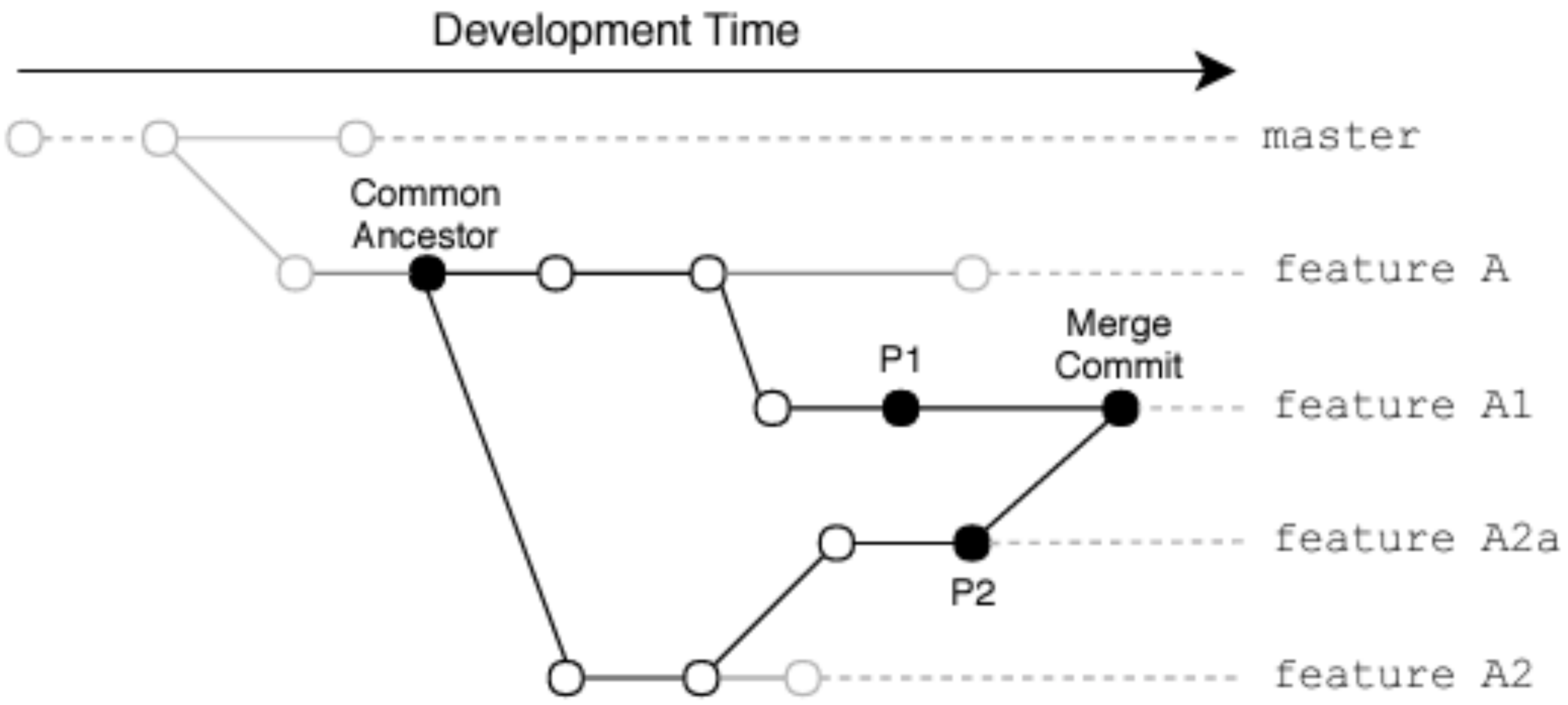
- Offer practices that facilitate collaborative software development

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”

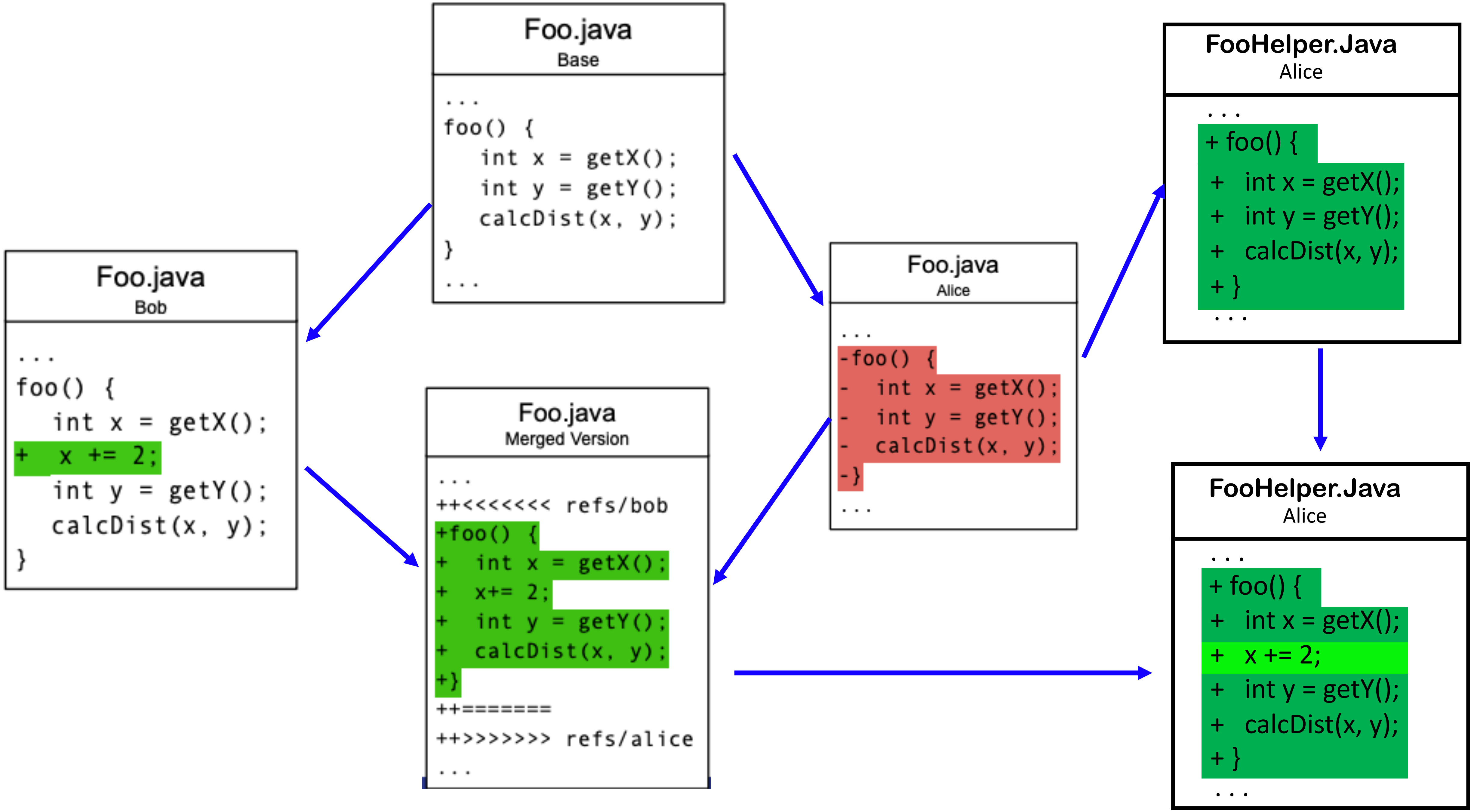
Merge Conflict

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() {
   int x = getX();
   int y = getY();
   calcDist(x, y);
}
...
```

```
...
foo() {
   int x = getX();
+  x += 2;
   int y = getY();
   calcDist(x, y);
}
```

```
...
-foo() {
-   int x = getX();
-   int y = getY();
-   calcDist(x, y);
-}
...
```

```
...
++<<<<<<< refs/bob
+foo() {
+   int x = getX();
+   x+= 2;
+   int y = getY();
+   calcDist(x, y);
+}
++=======
++>>>>>>> refs/alice
...
```

```
...
+ foo() {
+   int x = getX();
+   int y = getY();
+   calcDist(x, y);
+ }
...
```

```
...
+ 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

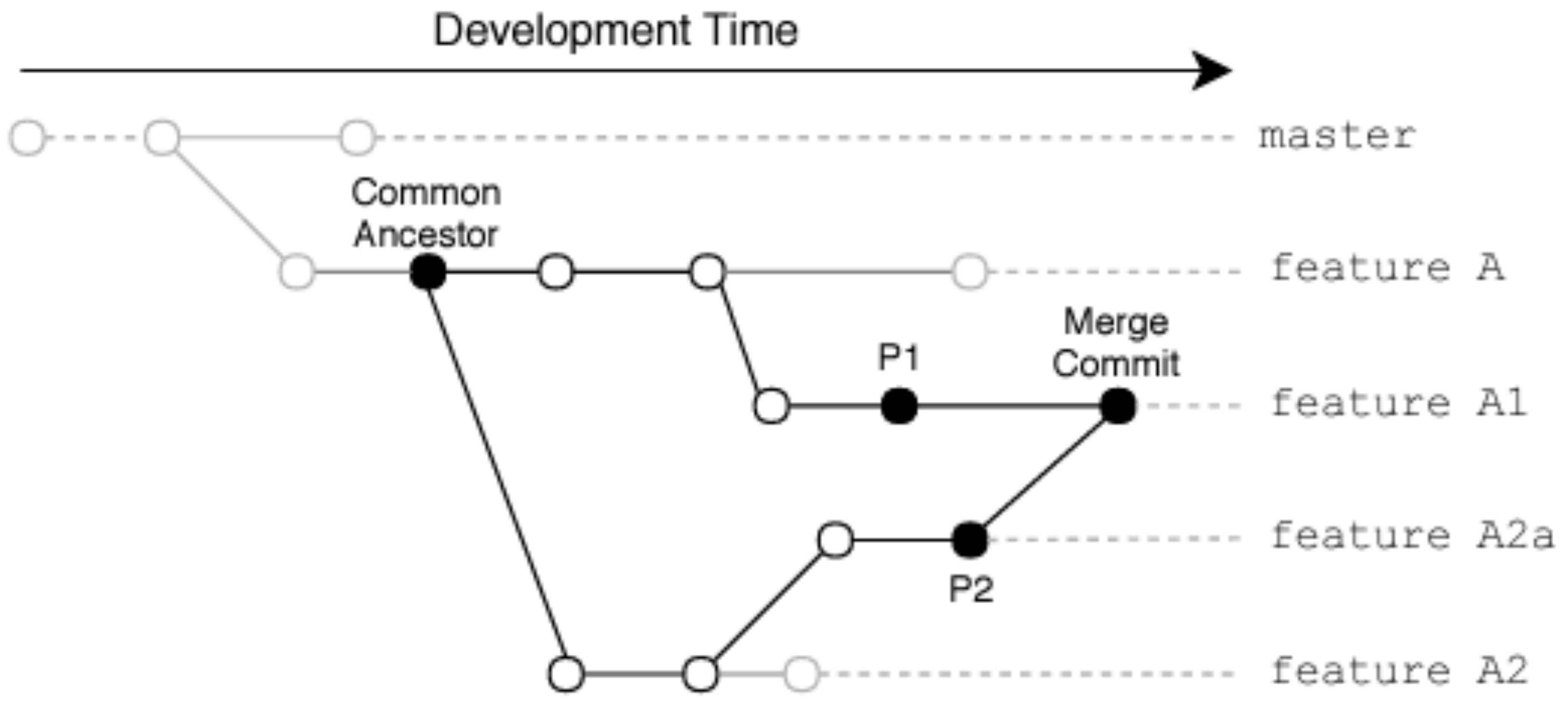
3. Detect Refactorings

4. Detect involved refactorings in the conflicting region

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

Ellis et al. A Systematic Comparison of Two Refactoring-aware Merging Techniques (2022). [https://arxiv.org/pdf/2112.10370.pdf]

Merge Scenario

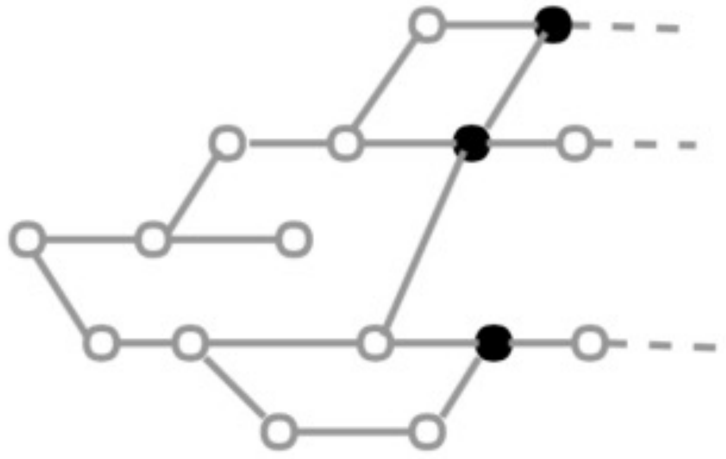


Step 1. Detecting Conflicting Regions

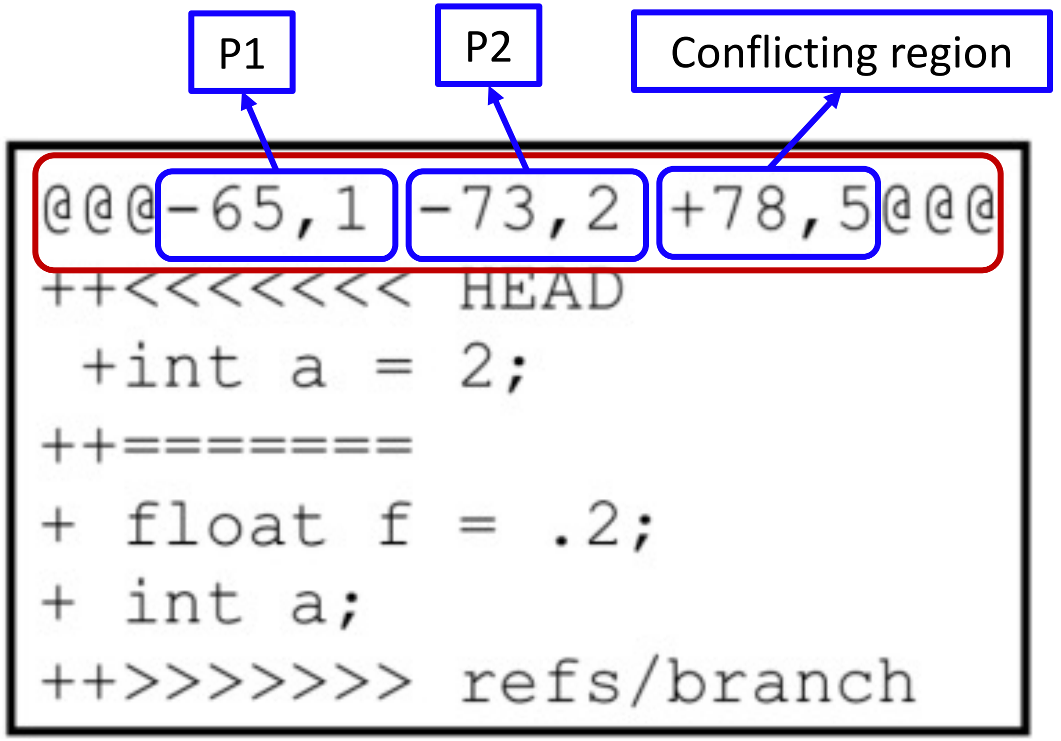
- First detect all merge scenarios
- Replay the merge scenario using the following commands:
  - git checkout P1
  - git merge P2
- If conflicting, git diff will output all conflicting regions
- Record the information to the database

Step 1. Detecting Conflicting Regions

Find merge commits



Find conflicting regions



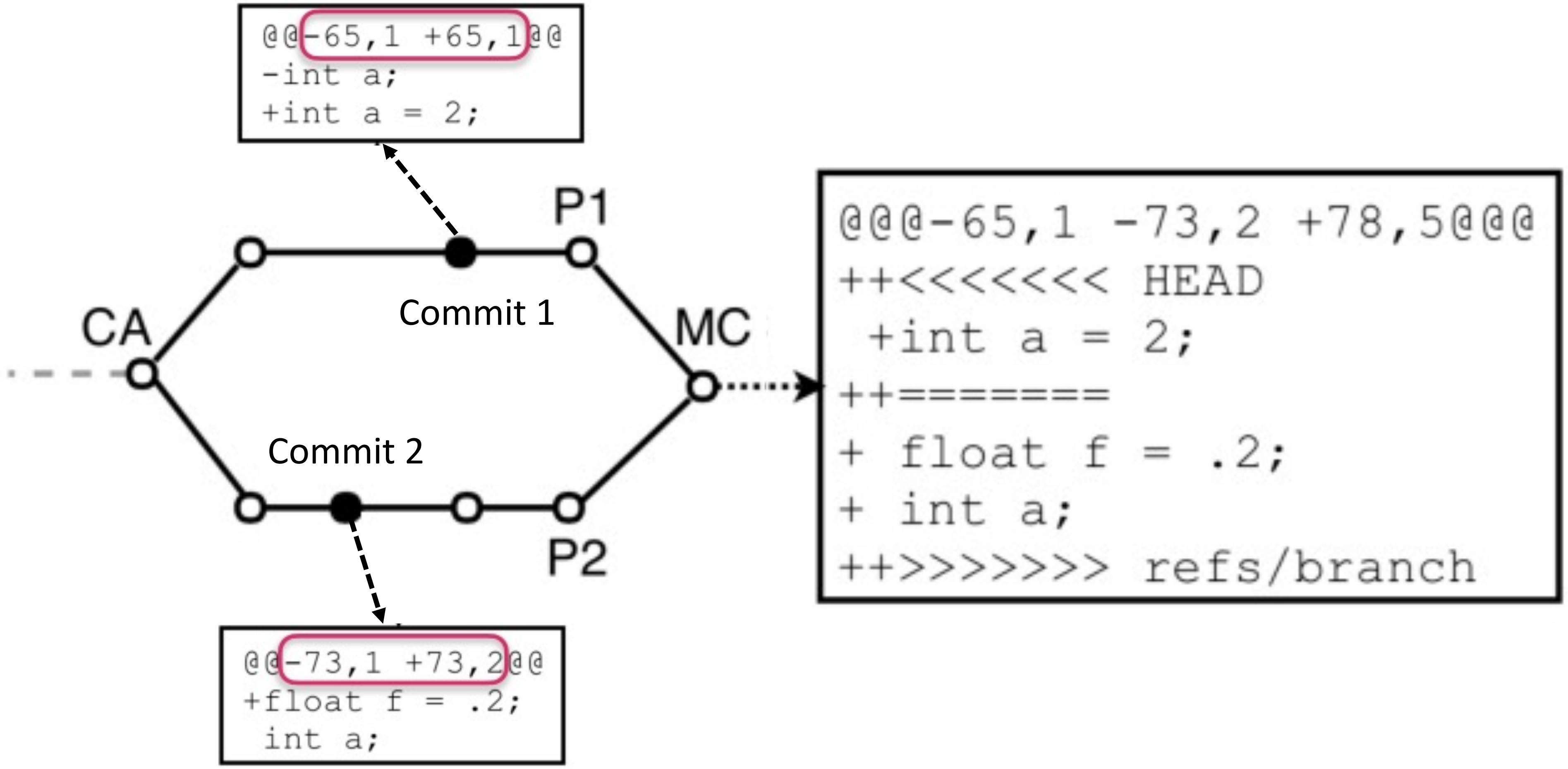
```
@@@ -65,1 -73,2 +78,5 @@@
++<<<<<<< HEAD
 +int a = 2;
++=======
+ float f = .2;
+ int a;
++>>>>>>> refs/branch
```

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

Step 2. Detect evolutionary changes

- For each conflicting region, we detect all commits that have touched that region
- These commits are called evolutionary commits
- Use the following commands:
    - `git log -L startP1,endP1:file P2..P1`
    - `git log -L startP2,endP2:file P1..P2`

Step 2. Detect evolutionary changes



```
@@-65,1 +65,1@@
-int a;
+int a = 2;
```

```
@@-73,1 +73,2@@
+float f = .2;
 int a;
```

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

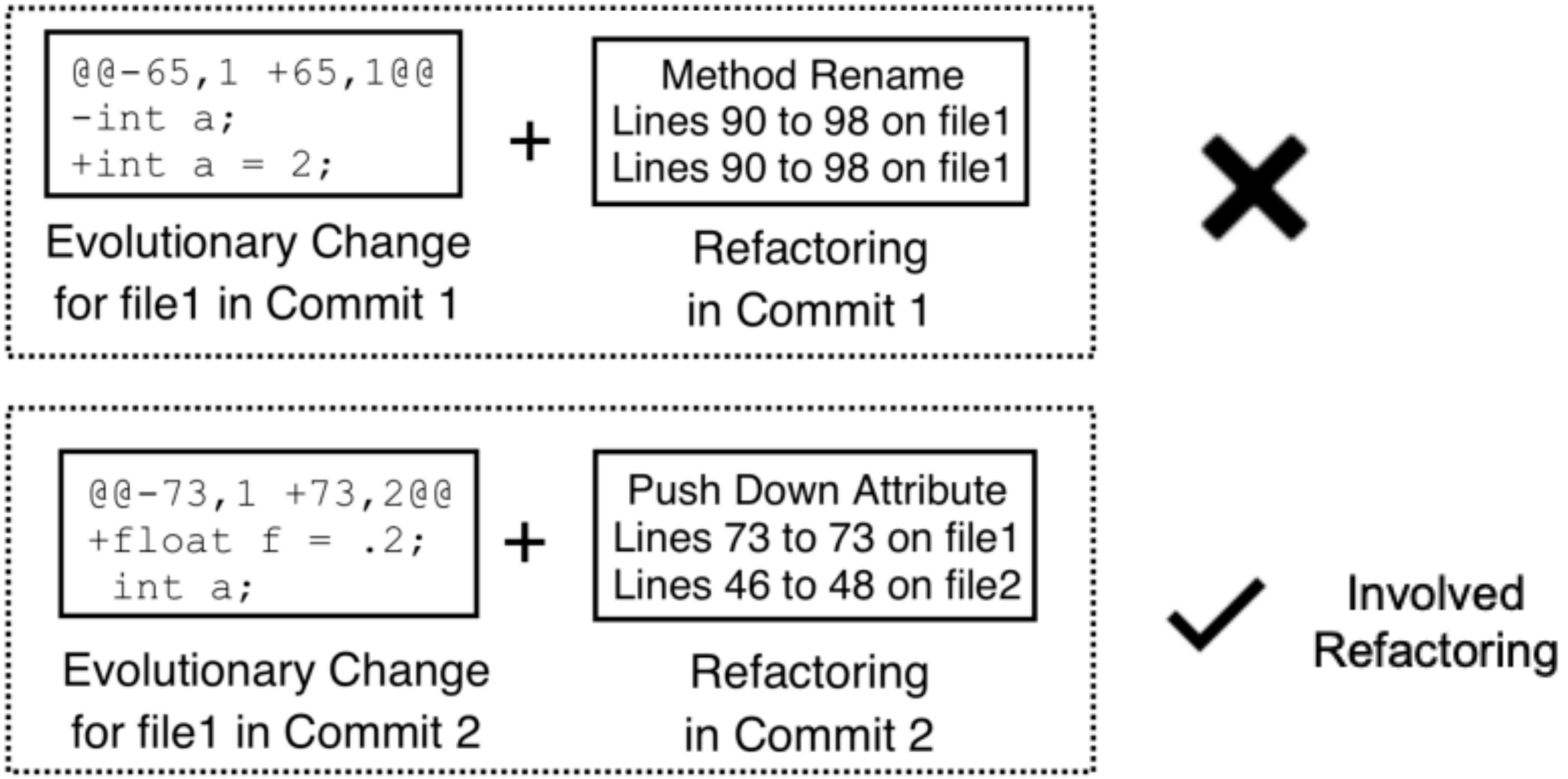
Step 3. Detect Refactorings

- RefactoringMiner is used in all the evolutionary commits
- We record the types and code regions of each refactoring to the database

Step 4. Detect involved refactorings

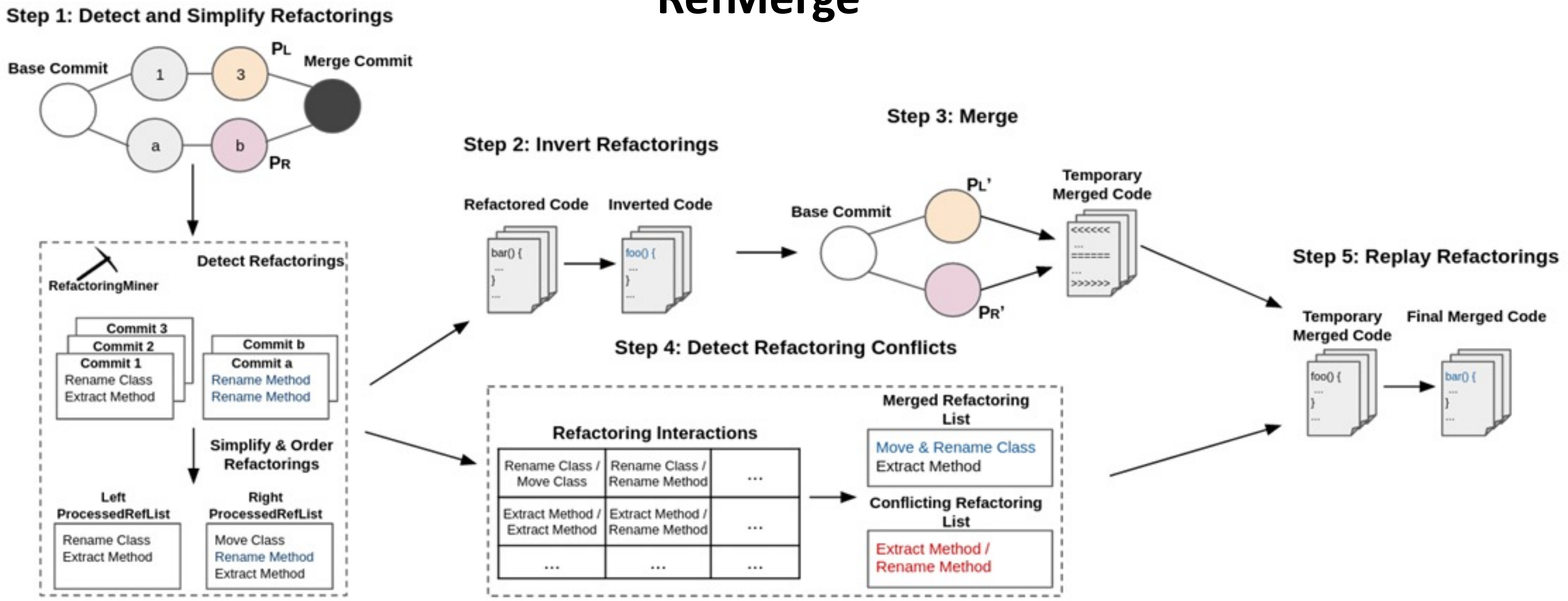
- The code range information for refactorings and evolutionary changes is used
- If there is an intersection between a refactoring and a refactoring, we call that refactoring an involved refactoring.

Step 4. Detect involved refactorings



Refactoring-Aware tools

RefMerge



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

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

The Project

- You will use **RefMerge** with the **git cherry-pick** instead of **git merge**.

- You will employ the same approach of mining git log to extract data that is interesting for the project.

- Let us go to the Lab