## MANUAL ANALYSIS PROTOCOL FOR DEFINING THE GROUND TRUTH

To define the ground truth, a double-check procedure must be performed:

- For each scenario, a pair of reviewers is assigned to conduct the manual analysis following the instructions below.
- The manual analysis must be carried out individually by each collaborator assigned to the scenario.
- The individual analyses of the two collaborators must then be compared:
    - If they converge, the result must be adopted as the scenario's ground truth.
    - If they diverge, the collaborators must meet and discuss the scenario together in order to reach a consensus.
        - If consensus is not possible, the scenario must be submitted to a third collaborator, who will make the final decision.

## INSTRUCTIONS FOR PERFORMING THE MANUAL ANALYSIS TO DEFINE THE GROUND TRUTH

The dataset consists of a number of subjects or cases to be analyzed.
Each subject is a pair composed of a merge scenario (a tuple with base, left, right, and merge commits) and a method or field declaration (yes, we also consider fields, not only methods) that was independently modified by two branches (or two developers).

A scenario that contains only a single declaration modified by two branches results in a single subject (and, consequently, a single row in the dataset spreadsheet).
A scenario *s* that contains *n* declarations modified by two branches results in *n* subjects:

*(s, declaration 1), …, (s, declaration n)*

Double check that you have the latest version of the dataset (https://github.com/spgroup/mergedataset) in your machine.

The dataset consists of a number of subjects, or cases to be analyzed. Each subject is a pair formed by a merge scenario (tuple with base, left, right, and merge commits) and a method or field (yes, we do not focus only on methods) declaration that was independently changed by two branches (or two developers). A scenario that contains just a single declaration that was changed by two branches leads to a single subject (and, consequently, a single row in the dataset spreadsheet). A scenario *s* that contains *n* declarations that were changed by two branches leads to *n* subjects: (*s*,declaration 1), …, (*s*,declaration *n*).

For each subject (*s,d*) in the dataset, the manual analysis should follow these steps:

1. Use kdiff to understand the changes made to base in the left and right branches
    a. Just load the base, left and right commits of *s* into kdiff (http://kdiff3.sourceforge.net) (kdiff3 left.java base.java right.java)
    b. Look for the declaration *d* that was changed in both branches
    c. Focus the analysis on this declaration
    d. We consider the left branch to be the sequence (path) of commits from the left commit up to base; similarly for the right branch
2. Take one or more screenshots that reveal the changes in the declaration that was the focus of the previous step, and how they are integrated into the merge commit
    a. Running the commands: kdiff3 left.java base.java right.java
    b. Add the screenshots to this presentation (http://gg.gg/imagesmerge)
    c. Considering the order in which the merge scenario appears in the dataset spreadsheet, add one slide with the merge commit hash
    d. Right after that, add one or more slides with the screenshots of the diff between left-base-right (this order is important)
    e. Following that, add a screenshot of the merge commit with "`// left`" and "`// right`" annotations at the end of the lines changed by the left and right branches, respectively
        i. If a line is changed by both left and right, add "`// left and right`"
        ii. The focus of the screenshots should be on revealing the changes in the declaration, but if the declaration is short enough the screenshots should show the whole declaration
    f. If the changes in left and right lead to a merge conflict
        i. Please add a note ("Merge conflict") to the slide
        ii. If the files in the merge commit still have the conflict markers (that is, the conflicts were not resolved), try to create a new commit that removes the markers and incorporates the changes from both left and right in a logical way
            - If this is not possible, the subject should be discarded.

            - Otherwise, keep going with the merged files in the new merge commit you created to fix the conflicts, and add its screenshot to the presentation as in item "d" (the presentation will have screenshots for the original and the fixed merge)

   g. If the changes in the left and right branches are not incorporated in the merge commit (some of the changes might have been removed or adapted by the integrator for a number of reasons: resolve a merge conflict, resolve a detected build, test or production conflict, remove redundancy, etc.), or are incorporated but the merge commit has additional changes made by the integrator (that is, some changes do not come from the left and right branches)
      i. Please add a note ("Merge amended by the integrator") to the slide
      ii. Replay the merge (git checkout left-hash; git merge right-hash), and check that the result is different from what is stored in the original repository
      iii. The manual analysis should proceed considering both merge commits (the original one in the repository, and the replayed one), with annotated screenshots for both (use "`// integrator`" if a line was changed by the integrador; if a line was changed by left, right and the integrator, add the three names to the comment)
         - The analysis of the original one will help us understand whether there is locally observable interference caused by the changes made by left, right and the integrator (so possibly an interference that was not caused by the changes in left and right, but by the changes made by the integrator)
         - The analysis of the replayed one will help us to understand whether there is locally observable interference caused by the changes made by left and right (so certainly an original interference caused by left and right changes)

3. Summarize the changes made in the left branch as observed in kdiff
   a. The focus is on what kdiff shows, not on all changes made by commits in the left branch
   b. So if an early commit in the branch adds, for example, an assignment to variable x (say `x=1`), and a later commit removes that, we won't see that in Step 1 above; the summary should also not mention this change, as the focus is on the end result
   c. The summary should be expressed in terms of
      i. The performed actions: added, removed, changed
      ii. The syntactic changes made to program elements: class, field, constructor, method declarations, initialization blocks, statements, expressions, arguments and parameters, if and while conditions, then branch, else branch, method body, loop body, annotations, comments
      iii. As in the following styles: "Left removes the reference to the field *maxRetries*, in the string expression that is returned by method *toString*";

“Left changes the condition of an if statement in method *outerHtmlHead.* The then branch of the if calls method *preserveWhitespace,* which is now only called if the parameter node is not null and is an instance of Element”

   d. To decide what should go in the summary, think about a toy example that would replicate the essence of the differences in the original subject. The summary should then describe such essence, using the vocabulary in the item above.

4. Summarize the changes made in the right branch as observed in kdiff (similar to the previous item)
5. Now, analyzing the merge commit, check whether there might be a Direct Flow (DF) from the execution of the changes in left to the execution of the changes in right, or vice-versa
   a. We consider that there might be such a flow when the changes (additions and modifications, we do not consider the effect of removals on DF) in one of the branches might semantically (that is, its execution) involve a **writing operation** (assignment, file write operation, method call with side effect, etc.) to a **state element** (local variable, parameter, static or instance field, file, stream, expression in return statement, exception raised, etc. So including temporary state too) that is transitively associated with a **reading operation** (usage reference in an expression or argument, file reading operation, etc.) involved in the changes (additions and modifications) made by the other branch
   b. As DF analysis involves understanding the execution of code changed by left or right, we also have to analyze the behavior of methods and constructors called by the declaration under analysis. So, for example, the code "`x=0;\\left ... C.m(x);`" where the whole `C` is marked as right has a DF only if the body of `m` actually reads the value stored in `x`; so we have to analyze the body of m in this case. If the call to `m` is also marked as right (as in "`x=0;\\left ... C.m(x); \\ right`"), then we have a DF no matter how the body of `m` refers to its parameter, since the execution of the method call itself involves computing the values of all arguments (as Java is not a lazy language).
   c. We are interested in detecting the possibility of a DF, not a guarantee of DF. So in "`x=0;\\left ... if (cond) C.m(x); \\ right`" we consider that there might be a DF, even though we are not sure whether `cond` can be eventually true. If we are sure that `cond` is always false, we don't say that there might be DF in the considered subject.
   d. The analysis is **local**, in the sense that we consider only the code of the declaration under analysis and of the methods and constructors that are called in its body. We do not consider the **global** surrounding context formed by the parts of the system that call the method or field declared by the declaration under analysis.
   e. Similarly, the integrated changes might affect different state elements (say window.size and window.color) that lead to a single combined visual effect on the system GUI, which is a (intended or unintended) globally observable interference. However, as our analysis is local, we focus only on the effects on the state

elements described before, which does not include the GUI visual state that is rendered by some GUI framework that accesses the non visual state elements (say window.size and window.color) that were changed. As the analysis is local, we don't even know if the changed state elements will be considered by such a framework.

f. For the reasons above, we consider only the effects related to the semantics of Java. So we consider that changes involving annotations have no semantic impact, as analysing such impact would typically require a global analysis for understanding the behavior of tools and frameworks that explore the annotations.
g. As DF must involve at least a **writing operation** and a **reading operation**, if the changes in both branches involve only read operations, or involve only write operations, or are only whitespace changes, you can quickly know that there is no DF
h. To establish if there is a DF, do not think about the current implementation of DF, simply follow the conceptual definition of DF as in the previous items
i. Think of the system state as the contents of its RAM, file system, input and output streams, what we can see in its display, etc. Think of a state element as a part of that, like a specific value stored in one of the RAM addresses. Assuming that, we consider "`a[6]`" refers to address 123, for example, of the RAM. Executing "`a[6] = 0`" changes this state element, changing the value stored in the RAM address 123. When we then execute "`b=a;b[6]=5`" we again affect what is stored in RAM address 123, that is, we affect the same state element as before. When we execute "`b=revert(a);a=b;a[6]=4`" we are now affecting another state element, another RAM address, assuming that revert returns a new array.
j. To analyze cases that have a line or statement marked as left and right (which could be caused by both branches making the exact same change in a line, or the integrator resolving a merge conflict by incorporating changes from both branches in the same line, or simply amending the merge commit for other reasons)
    i. Check first whether you could imagine equivalent code that uses auxiliary variables to keep changes from left and right in different lines and statements.
    ii. For example, a statement such as `variableFromRight.methodCallFromRight(new ObjectCreationFromBase().methodCallFromLeft())` (marked as both left and right) is equivalent to `x=new ObjectCreationFromBase().methodCallFromLeft(); variableFromRight.methodCallFromRight(x)`, in which the first statement would be marked as left, and the second would be marked as right, simplifying the analysis.

iii. Similarly, `variableFromRight.methodCallFromRight(new ObjectCreationFromBase(argumentAddedByLeft))` is equivalent to `x=new ObjectCreationFromBase(argumentAddedByLeft); variableFromRight.methodCallFromRight(x)`

6. Again analyzing the merge commit, check whether there might be a Confluent Flow (CF) from the execution of the changes in left and right to the execution of a common statement that comes from base
   a. We consider that there might be such a flow when the changes (additions and modifications) in each branch might semantically (that is, its execution) involve **writing** operations to different state elements that are transitively associated with **reading** operations executed by a statement that comes from base
   b. Similarly to what explained for DF, CF analysis involves understanding the execution of code, we might also have to analyze the behavior of methods and constructors called by the declaration under analysis. So, for example, the code "`x=0;\\left ... y=0;\\right this.m();`" requires the analysis of the body of `m` to understand if it has a confluence point, that is, a statement that comes from base and reads both `x` and `y`. Contrasting, in the code "`x=0;\\left ... y=0;\\right this.m(x,y);`" we can assert that there is CF because the method call is the confluence point, as both `x` and `y` are read when executing the method call.
   c. To establish if there is a CF, do not think about the current implementation of CF, simply follow the conceptual definition of CF as in the previous items
   d. Please check the other comments made for DF as they might apply here
7. Again analyzing the merge commit, check whether the execution of the changes in left might Override an Assignment (OA) from the execution of the changes in right, or vice-versa
   a. We consider that there might be such a flow when the changes (additions and modifications) in one of the branches might semantically (that is, its execution) involve a **writing** operation to a state element that is also associated with a **writing** operation involved in the changes (additions and modifications) made by the other branch **AND THERE IS NO WRITE OPERATION FROM BASE IN BETWEEN THEM!!!**
   b. To establish if there is OA, do not think about the current implementation of OA, simply follow the conceptual definition of OA as in the previous items
   c. Field references like `o.a` and `o.b` are considered different state elements and can be written by the branches without leading to OA. On the other hand, `o` is also a different state element, but cannot be changed together with `o.a` or `o.b` without causing interference; but they (`o` and `o.a`) can be written by both branches without leading to OA.
   d. CHECK IF THE SAME APPLIES FOR ARRAYS, MATHEUS
   e. Please check the other comments made for DF and CF as they might apply here

8. Finally analyzing the merge commit, check whether there is locally observable interference. There is no need to answer all the questions below; answering just one is enough.
    a. The changes in one of the branches interfere (negatively affect) with the changes in the other branch?
        i. Is there a state element x that left (right) computes a different value than base and merge? or
        ii. left, right and base compute the same value for x, but merge computes a different value?
    b. Does the behavior of the integrated changes (in the merge commit) preserve the intended behavior of the individual changes (in the left and right branches)?
        i. Do the new effects of left (right) in relation to base are preserved in merge, which has no new effects besides the ones in left and right?
    c. Can you think of a specification (pre-post condition pair) that is satisfied by the declaration in left (or right) but is not satisfied by the declaration in the merge? The specification should refer only to the state elements that are transitively involved in the changes in left (or right).
    d. Can you think of a test that would pass in left (or right) but fail in base and merge? The test assertion should refer only to the state elements (which might involve exceptions, as explained earlier) that are transitively involved in the changes in left (or right).
    e. Can you think of a test that would fail in left (or right) but passes in base and merge? The test assertion should refer only to the state elements that are transitively involved in the changes in left (or right).
    f. Field references like `o.a` and `o.b` are considered different state elements and can be changed by the branches without interference; `o` is also a different state element, but cannot be changed together with `o.a` or `o.b` without causing interference. The same applies for `a[0]` and `a[1]` (instead of `o.a` and `o.b`), and `a` (instead of `o`). The underlying reasoning is that `o.a` can be changed without changing or affecting the value stored in `o.b`, whereas when we change `o` we automatically affect the value stored on `o.a`. Contrasting, reading `o`, as in `this.m(o)`, not necessarily means reading `o.a`, `o.b` and all the fields of `o`. Elements of primitive types, strings, files, and streams are considered atomic; that is, changing different bits of the same integer is affecting the same state element, and so is changing different parts of the same text file, or writing to the same stream.
    g. If the changes in at least one of the branches affect a class signature (renaming of the class or its members; addition, removal, or visibility changes of class members), the previous two test related questions should assume that the signatures can be made the same in all commits without affecting behavior.
    h. If the changes in one of the branches are simply whitespace changes, or changes in comments, there is clearly no interference.

i. If the changes in one of the branches are simply structural (extract field, method, class, renaming, etc.) refactorings, and the resulting merge is valid (syntax and static semantics), make sure that they actually do not change behavior, and we can say that there is no interference.
    i. Changing "`if (x==null) return; y=x; if (x!=null) y=x.n();`" into "`if (x==null) return; y=x; y=x.n();`" is a refactoring but is not structural, and so can lead to interference. So we cannot say that there is no interference without analysing the changes in the other branch.
    ii. Changing the interface of a class C is not a refactoring of C. For example, changing method m of C from "`void m(){this.x=1;}`" to "`int m(){this.x=1;return this.x;}`" is not a refactoring of C, as this breaks clients of C. But this change in the declaration of m, together with changes that fix all clients of m, is a refactoring of the whole system if the clients are properly fixed in a way that preserves behavior. So if a class D that calls m as in "`c.m();r=c.x`" is changed to have "`r=c.m()`" then we can say that D was refactored (even if C was not). If D is the class under analysis, we can consider that it was structurally refactored, and so there is no locally observable interference.

j. If a field (instead of a method) declaration was changed in both branches, the analysis is easier: there is interference only if both branches change the initialization part of the declaration with different values. It's equivalent to the two branches changing the same assignment statement in the same method body.
    i. In these cases, there is no need to fill the DF, CF and OA columns, as we don't even consider that the analysis could be called in such situations (the mining framework doesn't invoke the analysis for these scenarios)

k. In many, but not all, situations the possibility of a DF, CF or OA does not imply on locally observable interference for a number of reasons:
    i. the possibility of DF, CF or OA is never realized due, for example, to a condition that is never true;
    ii. the detected DF, CF or OA already existed in base, and appeared in our analysis because the changes in the lines involved in the analysis had only whitespace changes or structural refactorings;
    iii. the detected DF, CF or OA did not exist in base, but has no harming effect (for example, when the changes in left and right assign the same value to the same variable; we have OA, but no actual interference).
    iv. That's why we have to answer one of the questions above, instead of simply checking if the analysis was positive in the previous items of this list.

l. In many, but not all, situations the lack of DF, CF or OA implies the lack of locally observable interference. This occurs especially because the analyses might not be sound (they might lead to false negatives, as when the code uses reflection, for example), and because interference could be associated with other kinds of analysis that we have not implemented (like an analysis that checks whether changes from one branch affect the control flow of the changes from the other branch).

   m. If one of the branches only removes code, then we can easily know that there is no DF, CF, or OA, but there could still be locally observable interference.

9. Check whether there might be a Control Dependence (CD) from the execution of the changes in left to the execution of the changes in right, or vice-versa.
   a. We consider that there might be such a dependence when the changes (additions and modifications, we do not consider the effect of removals on CD) in one of the branches might semantically (that is, its execution) preclude the execution of the changes (additions and modifications) made by the other branch.
      i. When there is a **left branch** in a conditional **CONDITION**, repeat command, or other statements and a **right branch** inside its body, where changes from the left preclude the execution of changes from the right, it can be executed differently and change the data flow, not executing the right changes.
      ii. When the method has more than one exit, it's necessary to analyze whether changes from the left causes a deviation to changes from the right, finishing in an exit and never executing your changes. Each exit is seen as a method finish flow, which can cause the execution flow to be interrupted and not execute the other lines of the method.
      iii. When a statement produces exceptions edges, if the left and right branches are within its scope, all statements can generate an exception and preclude each other's execution, since all statements generate an edge for the finish or change of the flow.

10. Fill the dataset spreadsheet with the information collected in the previous steps