# 1. [Optional] Rebase the source code repository:

* **Problem Description**: Especially Git projects contain a lot of branches that were merged into the master branch. However, the graph database storage assumes a linear history in which nodes and edges are tagged with version numbers:

A -> B ------> E -----> F -> G

|--- C ------- D --^

- File x.java is added from version B to version C

- In a linear version history: A -> B -> C -> E -> D -> F

– x.java must be removed from version C to version E

- x.java must be again added from version E to version D

* + *Problem:* This can result in many of such “branch switching” changes, which significantly increases the size of the history in the graph database.
  + *Solution:* We merge the changes into a linear history. In general, such a rebasing can result temporarily in conflicting changes if the same files are changed in different branches. To resolve the conflicts, we always keep the newest version of the file. The temporary conflicts are always resolved at the merge commit (F) in which the branches are merged in the history.
  + *Limitations:* The conflict resolution strategy alters the actual version history. As this seems to be an acceptable inaccuracy in most practical cases, an alternative solution would be to reorder the versions to minimize switching branches: A -> B -> E -> C -> D -> F. This adds the branch versions right before the merge commit (F).
* **Run Eclipse Application:**
  + *Implementation:* org.sidiff.bug.localization.dataset.GitRebaseApplication
  + *Launcher:* /plugins/org.sidiff.bug.localization.dataset/launch/RetrievalPhase00GitRebaseApplication.launch
  + *Program Arguments:*
    - **-dataset "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset.json"**
      * **The initial dataset configuration file with locations of the online code repository and bug tracker:**

{

"name": "org.eclipse.emf",

* Unique name of the dataset.

"repositoryHost": "https://git.eclipse.org",

"repositoryPath": "/r/emf/org.eclipse.emf.git",

* Online address of the Git repository: Host + Path

"bugtrackerHost": "https://bugs.eclipse.org",

* Online address of the bug tracker.

"bugtrackerProducts": ["EMF", "Modeling", "MDT", "MDT.XSD", "PDE"]

* The bug IDs are extracted by regular expressions from the commit messages. This can lead to false positive matchings of bug IDs. Therefore, we filter the bug reports by the given Products. Each report in the bug tracker is associated with a single product.

}

* + - **-sourcerepository "/users/mohrndorf/git/org.eclipse.emf"** 
      * **Path to the original repository. If the repository not exists, it will be cloned automatically.**
    - **-targetrepository "/users/mohrndorf/git/org.eclipse.emf\_rebased"**
      * **Path which will contain the rebased code repository.**
  + *Notes:*
    - Make sure the target repository folder does not exist.
    - If cloning the source repository failed, then delete the source repository folder (if one was created).
* **Finalize Result:**
  + The trace of the versions from source to the rebased target repository is saved in a new dataset file.
    - (The traces will be used in step 4. during reverse engineering the system model.)
    - <original dataset path>/<dataset name>\_dataset\_rebased\_<YYYY-MM-DD>.json

...

"history": {

"identification": "refs/heads/master",

"versions": [

{

"visible": true,

"identification": "8452787a6debf4377a1297ee32538f687d111ace",

* + - * + The Git version in the rebased target repository

"identificationTrace": "9984cee924e516631c4537729c47734f22e5948c",

* + - * + The Git version in the source repository

"date": "2021-07-06T07:22:06Z",

"author": "Pierre-Charles David",

"commitMessage": "[doc] Add missing top-level files"

},

...

* + To optimize disc space run garbage collection in rebased target repository:git gc –auto
  + In the context of this documentation, we will now replace the source repository folder with rebased target repository folder (remove \_rebased).
  + Save the rebased repository: <dataset name>\_dataset\_rebased\_<YYYY-MM-DD>.7z
    - It’s sufficient to save only the .git folder which stores the whole history.
    - Trace file: <dataset name>\_dataset\_rebased\_<YYYY-MM-DD>.json

# 2. Collect bug history from the bug tracker:

* **Problem Description**: The commit messages contain bug report IDs. However, the IDs are only stored textual by conventions like: [45625] fixed bug XY, bug fix 678768 for problem XY. After extracting these numbers by regular expressions, the IDs have to be looked up in the bug tacking system. The bug report will be stored in the dataset for the bug fixing commit, i.e., the committed version from which the ID was extracted. We also store the Git changes (file and line-based) of the bug fix commit as bug locations, and the Git file changes for incremental system model reverse engineering.
* **Run Eclipse Application:**
  + *Implementation and Configuration Provider:*
    - org.sidiff.bug.localization.dataset.DataSetRetrievalApplication
    - org.sidiff.bug.localization.dataset.retrieval.BugFixHistoryRetrieval
    - org.sidiff.bug.localization.dataset.retrieval.BugFixHistoryRetrievalProvider
  + *Launcher:* /plugins/org.sidiff.bug.localization.dataset/launch/RetrievalPhase01BugHistory.launch
  + *Program Arguments:*
    - **-dataset "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset.json"**
      * **The initial dataset configuration file with locations of the online code repository and bug tracker (same as in step 1.).**
    - -retrieval "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/configuration/RetrievalConfiguration.json"
      * The global dataset retrieval configuration:

{

"localRepositoryPath": "${user.home}/git/"

* + - * + The local folder for Git repositories. Project repositories will be cloned/opened to/from here by the dataset name.

}

* + - -bughistory
      * Run (only) the bug history collection step of the dataset retrieval application. In general, all steps can also be run successively. However, it’s (currently) recommended to “clear the runtime workspace” between the different steps. The launcher will ask for clearing.
* **Finalize Result:**
  + The application will print the bug reports/commit messages which were filtered or not found. If the result is not satisfying:
    - Add missing Products to the bugtrackerProducts in the initial dataset configuration.
    - Adjust the regular expressions which match the bug report IDs.
      * Configuration Object: org.sidiff.bug.localization.dataset.retrieval.BugFixHistoryRetrievalProvider
      * ID Matcher Implementation: org.sidiff.bug.localization.dataset.fixes.report.recovery.BugFixMessageIDMatcher
  + The bug reports and the version history is stored in a new dataset file:
    - <original dataset path>/<dataset name>\_dataset\_ bughistory\_<YYYY-MM-DD>.json

# 3. Retrieve the project history of the workspace:

* **Problem Description**: Eclipse workspace projects are added and removed during the history stored in the repository. In this processing step we will search all folders in the repository for Java projects and build an explicit history and storing their locations in the dataset file. The project nature (e.g. Java, PDE) can also be configured as filters in the WorkspaceHistoryRetrievalProvider configuration object. We can also filter projects by path patterns projectNameFilter, projectPathFilter in the dataset:

{

"name": "eclipse.jdt.core",

"repositoryHost": "https://git.eclipse.org",

"repositoryPath": "/r/jdt/eclipse.jdt.core.git",

"bugtrackerHost": "https://bugs.eclipse.org",

"bugtrackerProducts": ["JDT", "Platform"],

"projectNameFilter": ["converterJclMin", "converterJclMin1.5", "converterJclMin1.7",

"converterJclMin1.8", "converterJclMin9", "converterJclMin10",

"converterJclMin11", "converterJclMin12", "converterJclMin12",

"converterJclMin13", "converterJclMin14", "converterJclMin15"],

"projectPathFilter": ".\*?test.\*?|.\*?Test.\*?"

}

* **Run Eclipse Application:**
  + *Implementation and Configuration Provider:*
    - org.sidiff.bug.localization.dataset.DataSetRetrievalApplication
    - org.sidiff.bug.localization.dataset.retrieval.WorkspaceHistoryRetrieval
    - org.sidiff.bug.localization.dataset.retrieval.WorkspaceHistoryRetrievalProvider
  + *Launcher:* /plugins/org.sidiff.bug.localization.dataset/launch/RetrievalPhase02WorkspaceHistory.launch
  + *Program Arguments:*
    - **-dataset "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset\_bughistory\_2021-05-07.json"** 
      * **The bug history datafile that was created in the last step 2.**
    - -retrieval "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/configuration/RetrievalConfiguration.json"
      * The global dataset retrieval configuration (see above step 2.).
    - -workspacehistory
      * Run (only) the workspace project history collection step of the dataset retrieval application. In general, all steps can also be run successively. However, it’s (currently) recommended to “clear the runtime workspace” between the different steps. The launcher will ask for clearing.
* **Finalize Result:**
  + The workspace project history is stored in a new dataset file (including the previous information):
    - <original dataset path>/<dataset name>\_dataset\_ workspacehistory\_<YYYY-MM-DD>.json
  + Save the dataset: <dataset name>\_dataset\_<YYYY-MM-DD>.7z
    - <dataset name>\_dataset\_ workspacehistory\_<YYYY-MM-DD>.json
    - <dataset name>\_dataset\_ bughistory\_<YYYY-MM-DD>.json

# 4. Reverse engineering of the system model:

* **Problem Description**: The goal is to create a UML class diagram model which represents the whole project workspace (e.g. JDT) in a repository. Most existing techniques like MoDisco can only reverse engineer a single model per Eclipse Java workspace project. Moreover, the models can not be updated incrementally from the repository history, resulting in very bad runtimes. The invented tool can create a single model for a given workspace which contains multiple Java projects. The source code history is reverse engineered into a UML history which is also stored in a Git repository. Each Java-File has a corresponding UML-File. The UML-Files are combined by UML-Files which represent the package structure in a Java project, and a parent UML-File which represents the whole workspace. If a Java-File changes, we only update the corresponding UML-File. The references between the UML-Files are kept consistent by using the Java namespace to generate unique XMI-IDs for the model elements. References to external Java classes which are not in the workspace are represented by class stubs in a special library package of the UML model.
  + *Limitations:* We could\* currently not detect the following change for overloaded operation calls:

*( \*Currently operation calls are not reverse engineered.)*

class A {

void opA(B b) {

b.opB(“HelloWorld”)

}

}

class B {

opB(Object obj) {

System.out.println(obj)

}

}

Add overloaded operation to Class B:

class B {

void opB(Object obj) {

System.out.println(obj)

}

void opB(String text) {

System.out.println(“Text:” + text)

}

}

This changes the operation call b.opB(“HelloWorld”) to the newly added operation opB(String text) without changing class A. Such special cases could be traced in an operation call graph in the future in order to detect operation calls that are changed by adding overloaded operations.

* **Run Eclipse Application:**
  + *Implementation and Configuration Provider:*
    - org.sidiff.bug.localization.dataset.DataSetRetrievalApplication
    - org.sidiff.bug.localization.dataset.retrieval.WorkspaceHistoryRetrieval
    - org.sidiff.bug.localization.dataset.retrieval.WorkspaceHistoryRetrievalProvider
  + *Launcher:* /plugins/org.sidiff.bug.localization.dataset/launch/RetrievalPhase03SystemModelHistory.launch
  + *Program Arguments:*
    - **-dataset "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset\_workspacehistory\_2021-05-07.json"**
      * **The workspace history datafile that was created in the last step 3.**
    - **-datasettrace "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset\_rebased\_2021-05-07.json"** 
      * **The dataset containing the trace from the original source code repository to the rebased repository from step 1.**
    - -retrieval "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/configuration/RetrievalConfiguration.json"
      * The global dataset retrieval configuration (see above step 2.).
    - -systemmodelhistory
      * Run (only) the reverse engineering step of the dataset retrieval application. In general, all steps can also be run successively. However, it’s (currently) recommended to “clear the runtime workspace” between the different steps. The launcher will ask for clearing.
  + *VM Arguments:*
    - -Xms1000m -Xmx4000m
      * Requires a bit more runtime memory ;)
  + *Notes:*
    - Some exceptions result from older project configurations, e.g., “Manifest file is not present for…”. These warnings cannot be suppressed, but can be ignored in the context of the reverse engineering.
    - If projects are missing, check if the folder in the specific repository version contains a .project file with the required project nature (see above).
* **Resume the Eclipse Application:**
  + The reverse engineering creates checkpoints which can be resumed if the application fails for some reason.
  + Please follow the instructions in the code of org.sidiff.bug.localization.dataset.DataSetRetrievalApplication.
* **Finalize Result:**
  + The reverse engineered history is stored in a new dataset file (including the previous information):
    - <original dataset path>/<dataset name>\_dataset\_ systemmodel\_<YYYY-MM-DD>.json
  + The reverse engineered repository containing the UML history is stored in a new Git repository (beside the code repository):
    - <path to the common git base folder>/<dataset name>\_class
  + Save the dataset and repository: <dataset name>\_uml.class\_git\_<YYYY-MM-DD>.7z
    - It’s sufficient to save only the .git folder which stores the whole history.
    - Final Dataset: <dataset name>\_dataset\_ systemmodel\_<YYYY-MM-DD>.json

# 5. Converting the Git UML history into graph database history:

* **Problem Description**: In Git we would need to checkout each version of the UML history to create bug location samples. A checkout takes a few seconds; however, this can sum up to several hours if we need to check out 10.000 versions. Moreover, we would need to load the models (incrementally) from the repository storage for each version. To get fast access to samples of the model history, we convert the UML Git repository into a graph database. The graph database stores the model’s ASG. Each node and edge (relationship) is tagged with a version number. The version number is a range from the initial to the last version in which the node or edge exists in the model. If the attributes (properties) of a node changes, we mark the node as deleted (last version v) and create a new node (initial version v+1). A structural database query needs to filter the nodes and edges for a specific version by checking the version range.
* **Run Neo4j Database:**
  + *Compose Up:* /plugins/org.sidiff.bug.localization.prediction/docker/cip2/neo4j\_training/docker-compose.yml
    - VSCode (Docker View): context menu of file -> Compose Up
* **Run Eclipse Application:**
  + *Implementation:* org.sidiff.bug.localization.dataset.database.DatasetExportApplication
  + *Launcher:* /plugins/org.sidiff.bug.localization.dataset.database/launch/RetrievalPhase04DatabaseExport.launch
  + *Program Arguments:*
    - **-dataset "${project\_loc:/org.sidiff.bug.localization.dataset.domain.eclipse}/datasets/org.eclipse.emf/org.eclipse.emf\_dataset\_systemmodel\_2021-05-07.json"** 
      * **The dataset of the system model created in the last step 4.**
    - **-modelrepository "/users/mohrndorf/git/org.eclipse.emf\_class.uml"** 
      * **The Git repository of the system model created in the last step 4.**
    - -databaseconnection "bolt://localhost:7687"
      * The database connection protocol, host and port which is configured in the Docker compose file (ports).
    - -databasename "neo4j"
      * (Optional) The database name. Usually there is only one database per connection.
    - -databaseuser "neo4j"
      * The database user name which is configured in the Docker compose file (NEO4J\_AUTH).
    - -databasepassword "password"
      * The database user password which is configured in the Docker compose file (NEO4J\_AUTH).
  + *VM Arguments:*
    - -Xms1000m -Xmx4000m
      * Requires a bit more runtime memory ;)
* **Resume the Eclipse Application:**
  + The database conversion creates a single transaction for each version; therefore, the application can be resumed if it fails for some reason.
  + Please follow the instructions in the code of org.sidiff.bug.localization.dataset.database.DatasetExportApplication.
* **Finalize Result:**
  + Find <Volume-ID>, e.g., 8e91e60799d107e9b3f03aef13c78c12747fd7966a386c18cf58e9fe0fa12996:
    - Console: docker inspect neo4j\_training -> Mounts -> "Destination": "/data" (not /logs) -> Name
    - VSCode (Docker View): VOLUMES -> context menu of volume -> inspect -> "Destination": "/data" (not /logs) -> Name
  + Stop the container neo4j\_training
    - Console: docker stop neo4j\_training
    - VSCode (Docker View): CONTAINERS -> context menu of container -> stop
  + Insert volume ID and the dump file name, then run command to write a dump of the database:

docker run --interactive --tty --rm --publish=7474:7474 --publish=7687:7687 \

--volume="/var/lib/docker/volumes/<Volume-ID>/\_data/":/data \

--volume="/var/lib/docker/volumes/<Volume-ID>/\_data/backups":/backups neo4j:latest neo4j-admin dump \

--database=neo4j \

--to=/backups/neo4j-<dataset name>\_samples\_<YYYY-MM-DD>.dump

* + Save the database dump file from the docker volume: /var/lib/docker/volumes/<Volume-ID>/\_data/backups
  + Docker “Compose Down” and “Prune” volumes to save disc space.
    - VSCode (Docker View): context menu of file -> Compose Down
    - VSCode (Docker View): VOLUMES -> from toolbar -> Prune…

# Setup Neo4j Database

* **Database server setup (Version 4.2.5):**
  + Open in browser and set password
    - Login: neo4j, neo4j
    - <http://localhost:7474/browser/>
      * Troubleshooting: If page is not shown, delete browser cache (ctrl + F5).
  + Download >FULL< APOC Jar and copy it to plugins folder:
    - https://github.com/neo4j-contrib/neo4j-apoc-procedures/releases/
  + Settings in conf/neo4j.conf
    - dbms.security.procedures.unrestricted=apoc.\*
    - dbms.memory.heap.initial\_size=512m
    - dbms.memory.heap.max\_size=4G
    - dbms.allow\_upgrade=true
* **Open terminal (PowerShell) in the Neo4j database folder:**
  + Load database backup dump:
    - ./bin/neo4j-admin load.bat --from=./backups/<file name>.dump --database=neo4j --force
  + Run Neo4j database server:
    - ./bin/neo4j.bat console
  + Create Neo4j backup dump file:
    - ./bin/neo4j-admin.bat dump --database=neo4j --to=./backups/<file name>.dump

# 6. Splitting into training and test dataset:

* **Run the Neo4j database validation JUnit test to check the model consistency:**
  + org.sidiff.bug.localization.dataset.database.DatasetValidationTest
  + applyQuickFixes = true: Automatically fix some minor known issues, e.g., missing packages
  + Dump fixed database:
    - ./bin/neo4j-admin dump --database=neo4j --to=./backups/<file name>.dump
* **Run the Java application to compute the Neo4j splitting query:**
  + org.sidiff.bug.localization.dataset.database.DatasetSplitApplication
* **Apply the queries and backup and restore the database:**

# Setup Python Environment (with GPU support)

* Anaconda: [https://towardsdatascience.com/setting-up-tensorflow-gpu-with-cuda-and-anaconda-onwindows-2ee9c39b5c44]
  + conda create --name tf-gpu
  + conda activate tf-gpu
  + conda install python=3.8
  + conda install -c anaconda cudatoolkit=10.1
  + conda install pip
  + pip install tensorflow==2.3
  + pip install stellargraph
  + conda install nltk
  + pip install py2neo==4.2.5
    - Version must match the running database!

# 7. Training the Deep Learning Model

* **Start the Neo4j database with the training data:**
* **Compute the node self embedding dictionary:**
  + org.sidiff.bug.localization.prediction\src\node\_self\_embedding.py
    - src\buglocalization\textembedding\word\_to\_vector\_dictionary.py
      * pretrained\_dictionary\_name: Original Gensim Word2Vec model.
      * pretrained\_dictionary\_normalized\_name: Normalized Gensim Word2Vec model (fast loading, read-only).
      * pretrained\_dictionary\_feature\_size: Word embedding vector size.
    - src\buglocalization\selfembedding\dictionary\node\_self\_embedding\_word\_ranking.py
      * number\_of\_words: Limit the number of words
      * average\_word\_embedding: Average Of Word Embedding or Sum Of Word Embedding.
      * node\_signature: Compute node signature based on type.
      * node\_self\_embedding\_dictionary\_path: Path to store the node self embedding dictionary.
      * neo4j\_configuration: Connection information.
      * metamodel: Meta-model specific information
* **Setup configuration and run the training:**
  + src\bug\_localization\_training.py
    - doc\_description: Description for the information file.
    - neo4j\_configuration: Database connection information.
    - node\_self\_embedding: Path to the node self embedding dictionary.
    - training\_configuration: Deep learning model training hyperparameters and technical settings.
    - meta\_model: Meta-model specific information
    - model\_training\_base\_directory: Folder to store the trained DL model.
  + activate <environment>
  + python <project folder>/org.sidiff.bug.localization.prediction/src/bug\_localization\_training.py
  + Finally, store the trained DL model from <model\_training\_base\_directory>

# 8. Evaluating the Deep Learning Model

* **Start the Neo4j database with the test data:**
* **Setup configuration and run the test to generate the evaluation data:**
  + org.sidiff.bug.localization.prediction\src\bug\_localization\_prediction.py
    - evaluation\_results\_base\_path: Folder to store the computed predictions for each test sample.
    - bug\_localization\_model\_path: The trained DL model from <model\_training\_base\_directory>
    - prediction\_configuration: Technical evaluation settings.
    - neo4j\_configuration: Database connection information.
  + activate <environment>
  + python <project folder>/org.sidiff.bug.localization.prediction/src/bug\_localization\_prediction.py
  + Finally, store the predicted classifier rankings from <evaluation\_results\_base\_path>
* **Compute the diagram rankings from the original classifier rankings:**
  + src\evaluations\evaluation\_diagram\_ranking.py
    - evaluation\_results\_folder: The predicted classifier rankings from <evaluation\_results\_base\_path>
    - Run: Creates a new folder beside <evaluation\_results\_folder> with the derived diagram rankings.
* **Compute the evaluation metrics:**
  + src\evaluations\evaluation\_ranking\_metrics.py
    - evaluation\_results\_folders: The folders of the classifier and diagram rankings
    - Run: Creates a new folder “metrics” within each ranking folder with the CSV results.