Tool Support for Recurring Code Change Inspection with Deep Learning

Krishna Teja Ayinala

Faculty Mentor : **Dr. Myoungkyu Song**

April 5th 2019

Problems in Current Code Inspection Tools

- Developers need to inspect program differences on diff patches file by file
  - Despite recurring changes, a group of similar, related edits in multiple locations
- Manually examine individual edits
  - Tedious and error-prone process
- Current tools do not summarize recurring changes nor report anomalies in a diff patch
  - Hard to understand such code changes

RIDL: Recurring Code Change Inspection with Deep Learning

- RIDL: (i) summarizes recurring changes and (ii) detects potential change anomalies in the codebase
- Learn similar code fragments from a clone database to train a binary-class classifier
- Train the classifier by true and false clones, cloned and non-cloned pairs of code fragments
- Extract an edit script by analyzing data and control flow context
- Form change patterns by leveraging the classifier

Major Research Contribution

- Summarization for Recurring Changes
  - A novel integration of program differencing and AST-based code pattern search to track the changes to clones based on a deep learning technique.
- Detection for Change Anomalies
  - Detect potential change anomalies, such as missing and inconsistent edits.

Motivating Example

```
public class JEditTextArea {
   public void processKeyEvent(KeyEvent evt, int from) {
     Event event = (Event) evt;
-    if(inputHandler.isActive() && from != VK_CANCEL) {..
-      focusKeyTyped = event.getEvent();
-        ..
-     }
+ focusKeyTyped = processEvent(from, focusKeyTyped, event);
   ...
        event = (Event) evt;
-     if(inputHandler.isActive() && from != VK_CANCEL) {..
-        focusKeyTyped = event.getEvent();
-          ..
-    }
+ focusKeyPressed = processEvent(from, focusKeyTyped, event);
   }
 public void processFocusedKeyEvent(KeyEvent evt, int from) {
      Event event = (Event) evt;
-     if(inputHandler.isActive() && from != VK_CANCEL) {..
-         focusKeyTyped = event.getEvent();
-           ..
-      }
+ focusKeyTyped = processEvent(from, focusKeyTyped, event);
  case Event.KEY_PRESSED:
  event = (Event) evt;
-      if(inputHandler.isActive() && from != VK_CANCEL) {..
-         focusKeyTyped = event.getEvent();
-           ..
-      }
+  focusKeyTyped = processEvent(from, focusKeyTyped, event);
  ..}
  }
```

(a) Recurring changes for refactorings applied in revisions v20060919-7074 and v20060919-7075 in an open source project JEdit.

(b) Code fragments that have been missed to apply the required refactoring as (a).

(c) Code fragments refactored similarly but edited inconsistently unlike (a).

```
public class JEditEm
   void processKeyEv
int from) {
      Event focusKeyP
      Event event = (E
      switch(evt.getID
      case Event.KEY_PRESSED:
// It should apply a refactoring but missed required edits.
        if(inputHandler.isActive()
          && from != VK_CANCEL) {..
          focusKeyPressed = evt.getEvent();
           ..
        }
         ..
      }
   }
}
```

✓ RIDL summarizes these recurring changes

✓ RIDL identifies these change anomalies

```
xtArea {
nt(ActionEvent ev, int
tion;
n) ev;
       case Event.CTRL_MASK:
-    if(inputHandler.isActive()
-        && from != VK_CANCEL) {..
-         changeEventAction = ev.getEvent();
-             ..
-     }
// It should call to processEvent with three parameters, instead of calling to another overloaded method.
+      focusKeyPressedEvt = processEvent(from,
                        changeEventAction);
         ..
     }
}}
```

Overview of RIDL's workflow

- ✓ Phase 1. Feature extraction for change patterns
- ✓ Phase 2. Differencing and dependence analysis
- ✓ Phase 3. Recurring change summarization and change anomaly detection with trained models

Phase I. Feature Extraction for Change Patterns

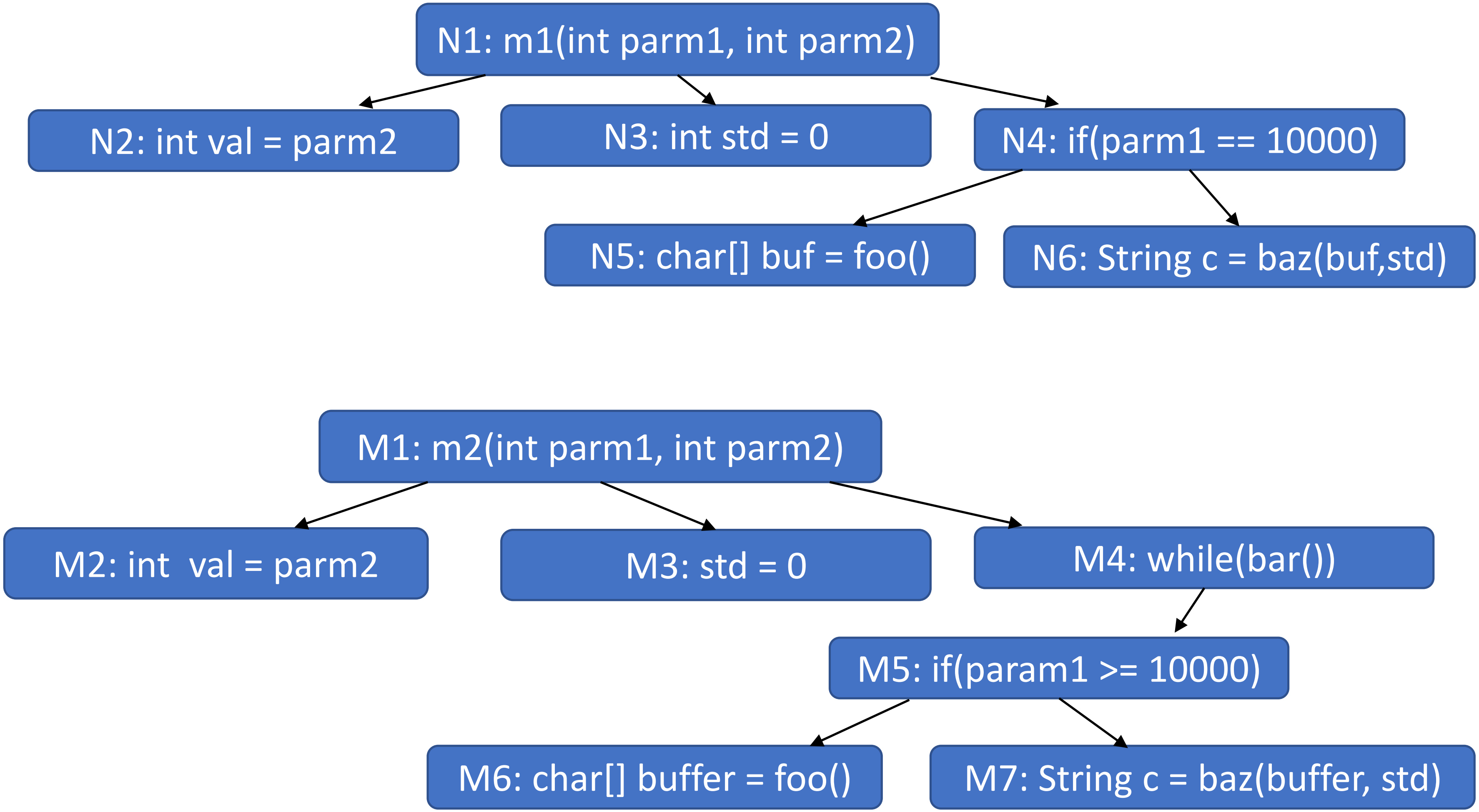
- AST Tree Matching
  - Code clones are converted to Abstract Syntax Tree (AST) model.
  - RIDL applies an efficient and worst-case optimal tree matching algorithm**, Robust Tree Edit Distance** (RTED).
- AST Node Categorization
  - **Five groups**: loop, exception, condition, declaration, control statements.
- Identifier Similarity
  - Identifier Normalization.
  - RIDL computes features using similarity scores of tokens.
  - RIDL computes the similarity score between the parameterized expressions by token level alignment using **Levenshtein Similarity calculation**.

Tree Matching Example

```
Public void m1( int parm1, int parm2) {
  int val = parm2;
  int std = 0;
  if (parm1 == 10000)  {
     char[] buf = foo ();
     String c = baz(buf, std);
  }
}
```

```
Public void m2( int parm1, int parm2) {
  int val = parm2;
  while(bar()) {
    std = 0;
    if (parm1 >= 10000)  {
       char[] buffer = foo ();
       String c = baz(buffer, std);
    }
  }
}
```

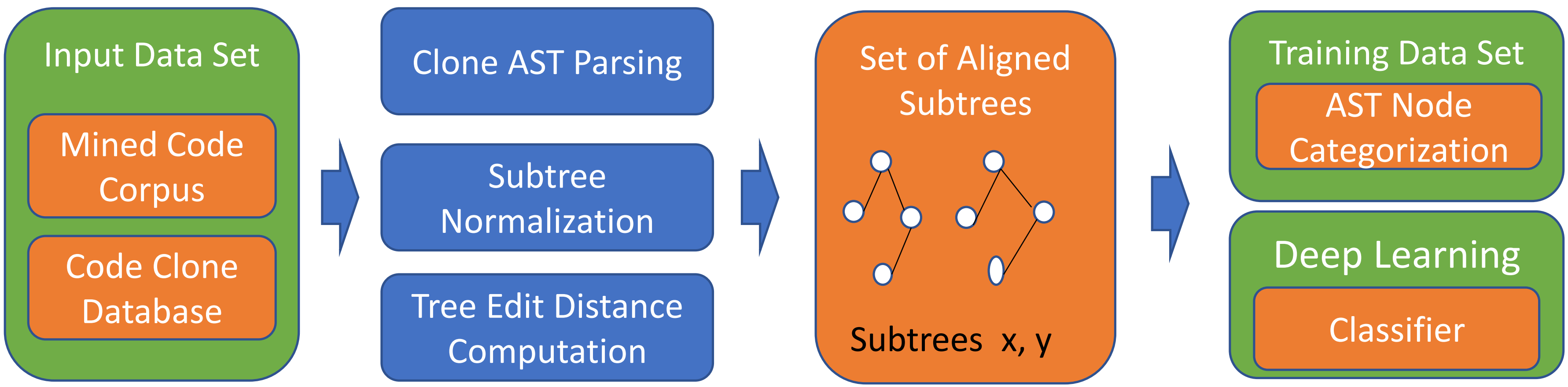
Tree Matching Example



Training

- Training a binary-class classifier to identify change patterns by characterizing the relationship of cloned (or non-cloned) AST subtree pairs
- The feature vectors of AST subtree pairs are extracted from cloned and non-cloned method pairs from a clone database
- Clone database is mined from 25,000 open source projects
- In the training data set, each data point forms <*node_type_vector*, *label*>
    - *node_type_vector* is a vector of five category scores (i.e., node alignment frequency and token similarity)
    - *label* is either 1 to denote cloned AST subtree pairs or 0 to non-cloned AST subtree pairs

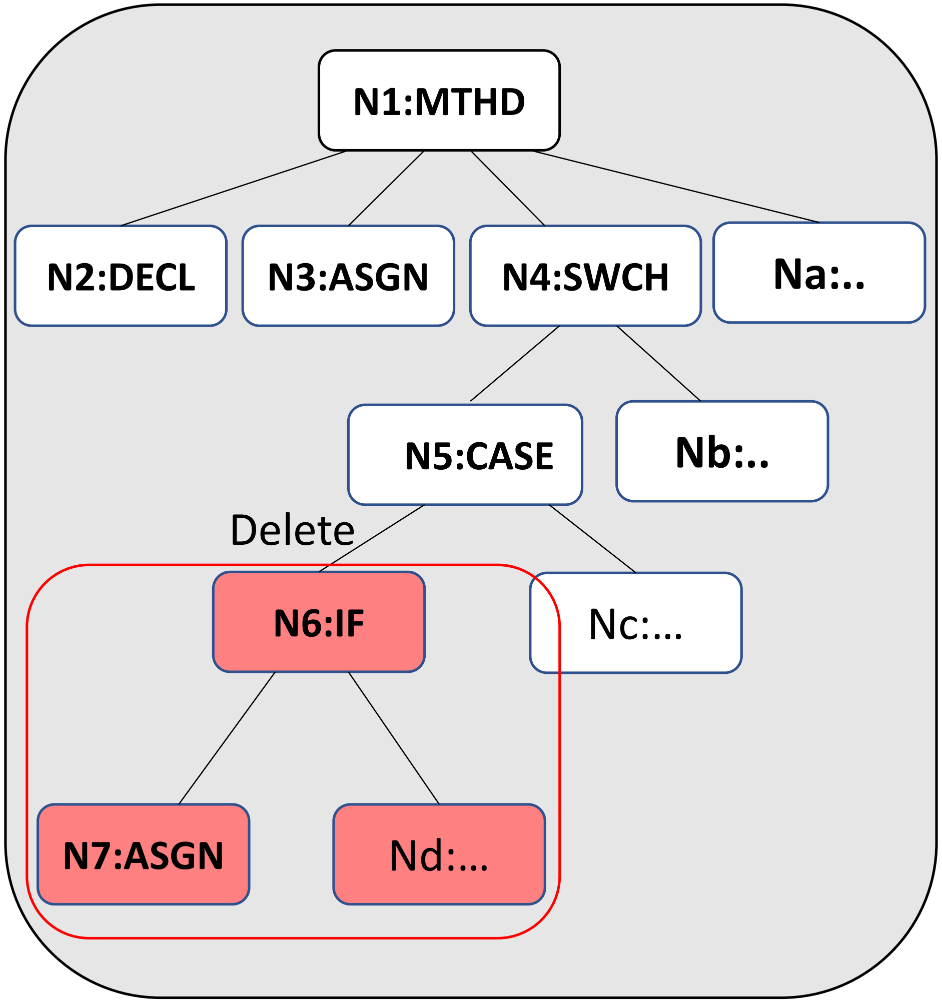
Extracting Features



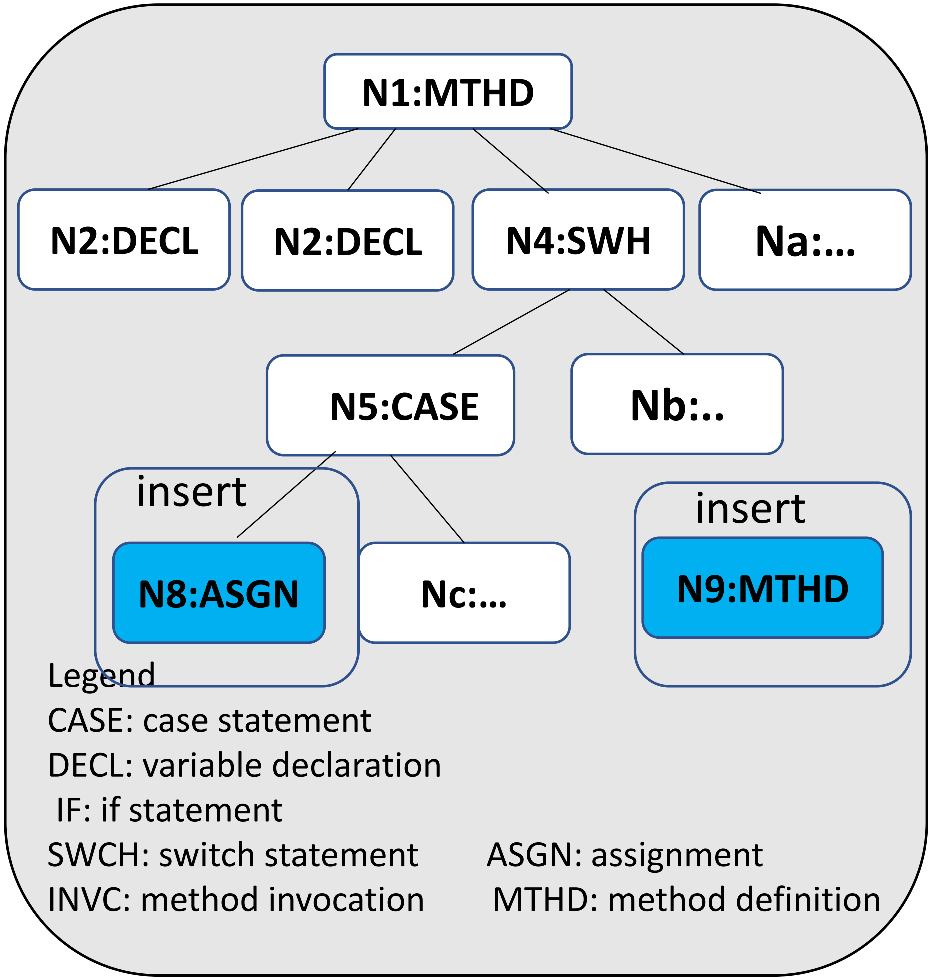
Phase II. Differencing and Dependence Analysis

- Computing AST differences
  - RIDL computes differences of original and edited versions of the program, using **AST differencing technique Change Distiller**.
  - Produce AST edit operations such as deletes, inserts, and updates
  - The extracted differences then are represented as **tree edit operations** to identify recurring changes
- Data and Control dependences
  - Data dependence: creating an edit script by analysing data dependences between edits and surrounding unchanged context.
  - Control dependence: creating an edit script by computing **control dependences** between the execution of edits and control predicates of unchanged context.

Analyzing Edits



(a) The AST subtree before edits

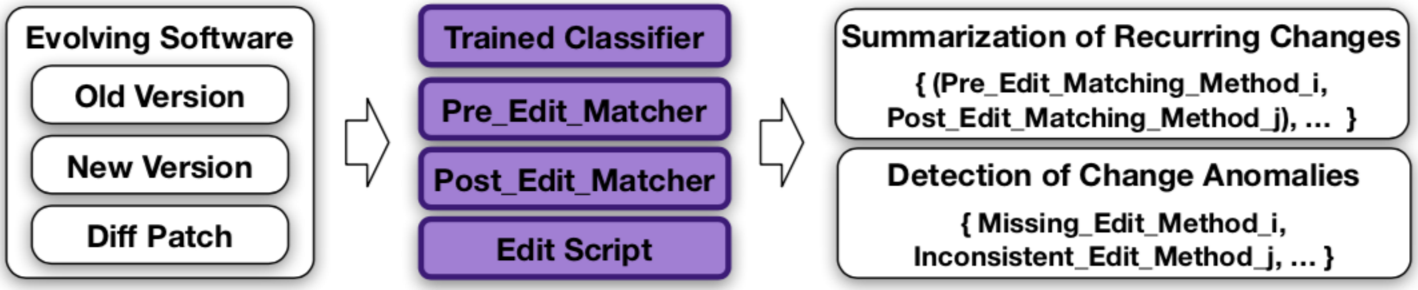


(b) The AST subtree after edits

Phase III. Recurring Change Summarization and Change Anomaly Detection with Trained Models

- Summarizing Recurring Changes
  - Using the edit scripts RIDL interoperates a pair of pre – and post-edit matchers such as ***Mpre*** and ***Mpost***.
  - *Mpre* and *Mpost* exploit the edit script to extract from a portion of a diff patch the categories of AST node types.
  - These matching implementations leverage the trained classifier to match specified changes against the code base to search for recurring change patterns.
- Detecting Change Anomalies
  - Detect a method that matches the pre-edit version but not the post-edit version or vice-versa.
  - Reporting missing or inconsistent edits

Summarizing recurring changes and detecting change anomalies with the trained classifier



[Tool Demo]

Evaluation

- The evaluation of the proposed approach aims to answer the two Research Questions
  - **RQ1. Can RIDL accurately identify similar code fragments using a deep neural network model?**
  - **RQ2. Can RIDL accurately summarize recurring changes and detect change anomalies?**
- The questions are answered by experiments with deep neural network models to evaluate the tool accuracy.

Experimental Design

- Similar Code Fragments Detection:
  - To evaluate our approach, we will apply RIDL to a clone database which has been mined from over 25,000 open source programs, including 2.3 million Java source files with over 365 MLOC.
- Change Summarization and Anomaly Detection:
  - To evaluate our approach in a real world setting, we collected the data set by manually examining clones and their changes where real developers applied recurring changes in Version Control System (VCS) repositories.

Dataset

Project	Files	Code
JFreeChart	1013	144703
Tomcat	2042	274785
JDT	1013	211718

Conclusion

- In this research, we propose a technique for inspecting recurring changes with deep learning.
- Our evaluation will show how accurately our static analysis approach with deep learning can effectively identify recurring changes and detect potential anomalies from open source projects.
- As future work for improving our approach and tool, we intend to (1) create pattern templates for more clone types; (2) provide tool support for fixing incomplete clone changes; and (3) implement checking operations to determine the correctness of extra edits to edited versions.

Thank You!!

- The research work is awarded with GRACA funds for 2019
- Questions and Answers