Implementation

For this project we have implemented a Clone Detector Tool. We have chosen to use TXL and Python for this implementation

Out approach is most appropriate to the one presented in NICAD clone detector. First we construct a parse tree of both the source file and the cloned one, then we search for clones at class, methods and blocks level.

TXL is a source code transformation programming language, a hybrid functional / rule-based language with unification, implied iteration and deep pattern match, according too TXL website, where it can be found more information about it. In TXL we have implemented the grammar parser and the tree matching at class and methods level. We used Python for implementing the Longest Common Subsequence matching, and for computing the percentage of similarity at the end. All the different components of our tool are connected by a C main program, which has the role of calling all the other components, and parsing the right arguments to them.

Our tool implements clones of type 1, type 2, type 3a and type 3b. Type one are the clones that are exactly matched, without normalization, and in continuous fragments of LOCs. Type 2 are the same as type 1, with the difference of identifiers normalization. Type 3 a accepts as clones blocks that are not continuous, allowing the operations of deletion and addition. This type does not normalize the identifiers. Type 3b is the same, with the difference that it normalizes the identifiers.

The parameters accepted by our tools are:

1. source file, according to which the tool finds clones
2. destination file, where the clones are searched
3. –type : 1, 2, 3a, or 3b, which is the type of clones matched
4. –threshold which represent the minimum number of LOCS which are matched as clones
5. –maxD : just for types 3a or 3b, which represents the maximum distance between 2 LOCs of the same clone

The only required parameters are the source file and the destination file. For the other there are some default values: type – 3b, threshold – 3, maxD – 6. According to these settings, the tool will output in the console the percentage of similarity and will output a file with all classes of clones. A class of clone consists in a class header, which represent the original source of the clone, with file name and line numbers, and in the actual list of clones. A clone can be either a class, interface, enumeration, method, or different blocks of code. Where it is the case, the tool will display also the operation of addition and deletion, with the corresponding LOCs (the original version’s LOCs for deletion).

A class is matched as a clone iff all its components are the same (identically or normalized). The same thing is for interfaces and methods. After this, the tool search into blocks of code, the ones that represent a clone, according to the tool’s settings.

This tool will not match fields of classes, import statements or packet header as clones, and will ignore brackets for all the future computations. For the computation of the percentage of similarity, all LOCs that are in at least one clone are counted. After this there are added the fields, import statement, packet headers, class and interface headers, which have a match in the original file. Since we do not take these components into account as clones, and since usually these will be just one LOC, we do not require any threshold for them. Just line matching is enough, for counting it in the percentage of similarity. This sum is divided with the total LOCs of the analyzed file, and this is the percentage of similarity.

Our tool has 6 main components: parser, pretty printer, parse tree matcher, LCS component, similarity counter, main.

1. Parser is implemented in the TXL. Because of the way in which TXL works, it automatically creates a parse tree of the source code of any file. For this it requires a grammar, which can be found on TXL website. This component is included in the files CloneDetector-type1.TXL and CloneDetector-type2.TXL. Once the parse is done, the subtrees are sent to the next component for clone matching.
2. Parse Tree Matcher is also implemented in TXL, and is part of the files CloneDetector-type1.TXL and CloneDetector-type2.TXL. It first extracts all the classes, interfaces and enums from the program, using TXL extract function (^). It will match every parse tree from the analyzed file with all parse trees from the source file. The matching is done according to the type of clone, either with instantiated identifiers, or with abstract identifiers. If there is an exact match, than the current component will be matched as a clone, will be removed from further analyzes, and will be outputted as a CloneOutput. The rest of classes and interface which are not matched goes to the next level of matching.

Here, all classes and interfaces are devided into their components: methods, constructers, or static initializers. As is mentioned above, fields are ignored. Here again is searched for a parse tree match, according to the type of clone, either with instantiated identifiers, or with abstract identifiers If it is not the case, then the next component, LCS, is called, for matching block level clones.

The normalization is done very easy, by just applying one TXL transformation rules, on the desired parse trees.

1. Pretty printer is an advantage of TXL. It automatically pretty prints the entire source code, according to the definition of grammar. We have printed every brackets on different line, for ignoring them in all the computation (threshold, maxD, similarity percentage). Also, all additional spaces or tabs are ignored, and every statement is printed on one line. This allows us to do a LCS matching on the ouputs provided by the TXL program, by ignoring all layout specification and brackets (which can be multiple and should not be considered as a line of a clone). Once this step is done, the pretty printed (and normalized version if it is the case), are parsed to the next component, for LCS matching.
2. The LCS matcher is a python script. It does the recursive method, creating the LCS matrix, and then returning the LCS according to this matrix. It will also output the bracketing from the cloned file, for keeping correct the grammar structure.

Since we are required to match multiple common sequences in the same methods, this script will do so. After every LCS is matched, the LOCs from it are removed from the tested method, and the LCS algorithm starts again. For displaying additions and deletion we keep an reference of additions and deletions next to the current line, for every line in the LCS. Additions and deletions prior to the sequence from LCS will not be outputted.

For normalized versions, the script takes as inputs the instantiated versions of source and cloned files too. In this case, even if the matching was done on the normalized version, in the output it will be printed the instantiated version. We think that this output may be more useful for the user of out tool.

1. The similarity counter is a simple python script, which receives as an input the entire sequence of LOCs included in all the clones, and the original version of the cloned file. Because of the pretty printer, we are able to do just a string based match. Also, because of the line numbering we can be sure that every line in these two files is unique, so there will be not false positives about this. For every line in the original source file matched with a line in the clones, a counter is increased. Brackets are ignored here too. After this, the counter is divided by the total number of LOCs from the original cloned file, and the percentage of similarity is computed.

At this step, at the LOCs from the clones, there are also added the fields, class definitions and interface definitions. Since these cannot be included in any clones (class and interface definitions are included just if the entire class/interface is a clone), and we want to output a percentage of similarity, we thought about including them here, by requiring just one match.

6) All these components are connected together by a C program. This will be called from command line, will take as parameters at least the two files names, will define some default values for the other parameters, and will call all these programs. It will output the percentage of similarity in the console, and the clone classes in the output file.