An evaluation of duplicate code detection using anti-unification

Peter Bulychev
Lomonosov Moscow State University, Russian Federation
peter.bulychev@gmail.com

Marius Minea
Politehnica University of Timişoara, Romania
marius@cs.upt.ro

***Abstract*—This paper describes an algorithm for finding software clones, which works at the level of abstract syntax trees and is thus conceptually independent of the source language of the analyzed programs. We use a notion of clones which captures replacement of subtrees in the program AST, and is formally based on the notion of anti-unification. This allows us to capture syntactic structural similarity with increased accuracy and express it in metrics. We have implemented this algorithm in a tool named Clone Digger, freely available under the GPL license, and which currently supports the Python, Java and Lua languages. We report on initial experimental results and comparisons with similar tools.**

I. Introduction

Duplicate code can occur as a result of approaches to development and maintenance, due to language or programmer limitations, or simply by accident [1]. Code duplication – also called cloning – can be a significant drawback, leading to bad design, and increased probability of bug occurrence and propagation. As a result, it can significantly increase maintenance cost (for instance, any bug in the original has to be fixed in all duplicates), and form a barrier for software evolution. Since the occurrence of cloning remains widespread, with reported amounts varying from 6.4%–7.5% to 13%–20% [1], duplicate code detectors are a valuable class of software analysis tools, and a first step in software clone management [2].

We have developed a new algorithm for detecting software clones. Our approach belongs to the class of methods based on abstract syntax trees. Our main goal is to define an accurate characterization of the *structural similarity* of two code fragments, on the basis of which they are classified as code duplicates. We formalize this using the concept of *anti-unifier*, which denotes the most specific generalization of two terms. Anti-unification was first described by Plotkin [3] and Reynolds [4]. Our approach works in several steps, first using anti-unification to calculate the distance between two abstract syntax trees corresponding to code fragments, and then groups similar trees into equivalence classes called clusters. Anti-unification naturally captures the *most specific* pattern which matches two similar abstract syntax trees, and thus directly provides a means to measure their structural differences. Moreover, since unification captures *sharing* of syntactical elements, clones in which shared subtrees are *consistently substituted* [2] with the same values are naturally classified by our approach as closer than inconsistent substitutions.

Overall, we consider two large enough sequences of statements as a clone if one of them can be obtained from the other by replacing some subtrees such that the total size of replaced subtrees is less than some given threshold. To extend anti-unification-based similarity from single statements to sequences of statements, we a use compound three-phase algorithm. First, we partition all statements into clusters using anti-unification distance; the result is an abstract view of the code as a sequence of cluster identifiers. Next, we find all pairs of identical subsequences of cluster IDs. Finally, we compare the resulting pairs of statement sequences based on an overall similarity metric. This check is again performed using anti-unification distance, and duplicates are reported if the distance is below a certain threshold.

Our method is closest to the fully syntactic abstraction approach developed in [5]. Their algorithm detects a similarity between, e.g., `a[1]` and `a[x+1]` by reducing them to the pattern `a[?]`. This pattern can be seen as anti-unifier of the two expressions. We extend the pattern-based comparison approach to more complex programming constructs: ultimately, our algorithm performs unification on sequences of candidate statements. Moreover we provide metrics to assess clone similarity, whose quality increases if the occurrences of the same variable (in the same scope) refer to the same leaf in the abstract syntax tree.

We have implemented our algorithm in the Clone Digger tool, available under the GPL license, written in Python and currently supporting Python, Java and Lua as input. It can be extended with parsing support for other languages.

This paper is a continuation of [6], where we have initially described the anti-unification based clone detection algorithm. Here, we focus on providing an evaluation of the CloneDigger tool on several open-source software projects.

II. Anti-unification

A. Definition

Anti-unification was first studied in [3], [4]. As its name suggests, it produces from two terms a more general one that covers both, rather than a more specific one as in unification.

Let E_1 and E_2 be two terms. Term E is a generalization of E_1 and E_2 if there exist two substitutions σ_1 and σ_2 such that $\sigma_1(E) = E_1$ and $\sigma_2(E) = E_2$. The most specific generalization of E_1 and E_2 is called anti-unifier. The process of finding an anti-unifier is called anti-unification.

Anti-unification was originally described for trees. We actually work with directed acyclic graphs, since leaves of an abstract syntax tree representing the same variable reference may be merged, but anti-unification can be extended in a straightforward way to this context. We use the anti-unification algorithm described in [7].

The anti-unifier tree of two trees T_1 and T_2 is obtained by replacing some subtrees in T_1 and T_2 by special nodes which contain term placeholders marked with integer labels. We represent such nodes as $?_n$. For example, the anti-unifier of $Add(Name(\texttt{i}), Name(\texttt{j}))$ and $Add(Name(\texttt{n}), Const(1))$ is $Add(Name(?_1), ?_2)$. In some abstract syntax tree representations, repeaded occurrences of the same variable refer to the same shared leaf. In this case, the anti-unifier of $Add(Name(\texttt{i}), Name(\texttt{i}))$ and $Add(Name(\texttt{j}), Name(\texttt{j}))$ is $Add(Name(?_1), Name(?_1))$.

B. Anti-unification features

The anti-unifier of two trees represents their common "skeleton", and contains placeholders for subtrees which differ. Anti-unification extends straightforwardly to sets: the anti-unifier of a set of trees is the most specific pattern which matches each tree in the set. It can therefore be viewed as a common characteristic of a set of trees. This anti-unification feature was used in [8] to discover widespread patterns of formulas in scientific articles.

An anti-unifier captures only the common top-level tree structure, up to the first mismatch on each branch top-down. For instance, the anti-unifier of the two trees $Add(Add(Name(\texttt{a}), Name(\texttt{b})), Name(\texttt{c}))$ and $Add(Name(\texttt{a}), Add(Name(\texttt{b}), Name(\texttt{c})))$ is $Add(?_1, ?_2)$. It captures the first-level similarity but lacks details on the second level, where subtrees have different structure.

Anti-unification leads to a natural notion of distance between two trees. Let U be the anti-unifier of two trees T_1 and T_2 with substitutions σ_1 and σ_2. Let n be the number of placeholders in U. Then σ_1 and σ_2 are mappings from the set $\{?_1, ?_2, \ldots, ?_n\}$ to substituting trees. Define the size of a tree as the number of its leaves. This notion of size is robust to the particularities of representing abstract syntax trees because it corresponds to the number of all name and constant occurrences. We now define the anti-unification distance between T_1 and T_2 as a sum of sizes of all substituting trees in σ_1 and σ_2, minus the number of placeholders.

For example, consider two trees $Add(Name(\texttt{i}), Name(\texttt{j}))$ and $Add(Name(\texttt{n}), Const(1))$. The anti-unification substitutions are σ_1=$\{\texttt{i}/?_1, Name(\texttt{j})/?_2\}$ and σ_2=$\{\texttt{n}/?_1, Const(1)/?_2\}$, the sizes of trees in the substitutions are $|\texttt{i}| = |\texttt{n}| = |Name(\texttt{j})| = |Const(1)| = 1$. Therefore the anti-unification distance for this example is 2.

Anti-unification distance can be seen as tree editing distance [9] with a restricted set of operations. It captures the structural differences between two trees and does not allow permutation of siblings or changing the number of child nodes.

III. Definition of clones

Our goal is to find duplicate fragments of code by discovering similarities between subtrees and sequences of subtrees in the program's abstract syntax tree. We use an AST-based method because the structural nature of these approaches generally leads to higher precision, and can later be more easily employed for corrective action. Abstract syntax trees also allow an analysis with flexible level of granularity: we can identify, for example, renamed identifiers or modified subexpressions.

We search for clones in sequences of statements; the smallest unit of duplicate code we report is a statement. We also handle definitions of classes and functions, which are similar cases (their bodies being essentially compound statements). We thus focus the presentation on statements, for simplicity.

We consider a pair of statement sequences to be clones if one of them can be obtained from the other by replacing some subtrees with other subtrees such that the total size of these subtrees is less than a certain threshold. (In addition, we require the statement sequences to exceed a given length threshold of interest). It is easily seen that the total size of the replaced trees is equal to the anti-unification distance between the considered code fragments.

This definition allows all structural substitutions, e.g., subexpressions can be replaced with arbitrary other subexpressions. According to the taxonomy presented in [2], our method handles inconsistently structure-substituted clones. In the current setting, we only treat contiguous clones, and do not allow clones with gaps, or inserted/deleted statements. This is due to the second step of our algorithm, which after clustering similar statements and abstracting statements to cluster IDs looks for *identical* sequences of *consecutive* IDs. It could be conceptually extended to sequences with gaps, by unifying arbitrary statements with the empty statement.

IV. Duplicate code detection algorithm

Following the approach of [10], [11], we first linearize the abstract syntax tree of the program. As a result, all sequences of statements and definitions are presented in the abstract tree as sibling subtrees.

To find *all* clones satisfying our definition one would need to compare every statement subsequence in the program with every other sequence. To obtain a practical method, we propose an approximation consisting of three phases:

1) Identify similar statements using anti-unification and partition them into clusters with the same anti-unifier. After the first phase each statement is marked with the ID of its cluster. For example, cluster 1, represented by the anti-unifier $?_1$+=$?_2$ might include the statements `i+=j` and `m+=2*n`, while $?_1$++, identified as cluster 2, includes statements `i++` and `j++`.
2) Find identical sequences of cluster IDs, corresponding to statement sequences within a compound statement. These statement sequences are candidates to be reported as code clones.

3) Refine by examining the candidate sequences identified previously for overall similarity. Anti-unification is used again to compute a similarity metric for each pair of candidate statement sequences.

We describe these phases in more details below.

A. Comparing and clustering similar statements

As discussed above, the anti-unifier of a set of statements can be viewed as its common skeleton. We implement a simple clustering algorithm based on anti-unification distance. Each newly examined statement is compared with the anti-unifiers of existing clusters. If one of them is sufficiently close, the statement is placed in that cluster (which potentially updates its unifier), otherwise, a new single-statement cluster is created. More details of the algorithm are described in [6].

B. Detecting pairs of identical cluster sequences

After the algorithm's first phase, each statement is marked with its cluster ID. In the second phase, we search for all pairs of statement sequences statements which are identically labeled. Only sequences with length above a certain threshold are considered. This search is performed using a suffix tree approach [12]. Detected pairs of statement sequences are clone candidates.

C. Checking overall similarity of code sequences

The second phase of algorithm produces a set of clone candidates. Suppose we have a candidate pair consisting of the statement sequences $\{s_1, s_2, \ldots, s_n\}$ and $\{t_1, t_2, \ldots, t_n\}$. To check this pair for similarity, we construct two new trees `Block(`$s_1, s_2, \ldots, s_n$`)` and `Block(`$t_1, t_2, \ldots, t_n$`)` and compare them using anti-unification distance. If the distance between them is below a certain threshold, then this pair is reported as a clone.

This overall filtering phase is necessary, since not all candidates found in phase two should be reported. Although the statements in the two sequences may pairwise have the same cluster IDs, the overall amount of their individual differences may be too great to meaningfully consider them as clones.

Yet, the overall distance between two sequences cannot be obtained by summing distances between corresponding statements; an overall view is needed. Consider the statement sequences `{i=0;i+=1;f(i);}` and `{j=0;j+=1;f(j);}`. The distance between each pair of corresponding statements is 1, for a sum of 3. However, this does not account for variable sharing. Anti-unification on the other hand returns the value 1, since a single overall substitution is required to map between the two sequences. Thus, our algorithm naturally classifies consistent substitutions as closer than inconsistent ones.

The algorithm has a worst-case quadratic complexity component, since in the first phase, anti-unification is attempted for each statement with each potentially matching cluster (i.e., with the same hash value for the *d-cap* representing the top-level tree structure, similar to [5]). This process is repeated after creating all clusters, to obtain the closest matches, since anti-unifiers change as clusters grow. Practical results show confirm that the first phase dominates the overall cost.

V. Comparison with existing approaches

There is a large body of work in the duplicate code detection field. An extensive survey is [1], while [13] does a detailed comparison of experimental results.

The approach closest to our work is by Evans et al. [5], which performs a fully structural abstraction (over arbitrary subtrees) rather than using lexical abstraction (which allows parameter substitution only, e.g. for identifiers or constants). For example, structural abstraction captures the similarity between `a[x]` and `a[y+1]` using the tree pattern `a[?]`. The algorithm of [5] works in bottom-up manner, increasing the size of the detected common patterns in the abstract syntax tree step-by-step, and uses several heuristics.

Anti-unifiers can also be viewed as patterns, but in addition they naturally capture shared subtrees (multiple equal substitutions) – in principle, patterns could be enriched for this purpose as well. In our approach, anti-unifiers are built in top-down manner by enlarging clusters and generalizing their anti-unifiers. The use of clusters is similar to the pioneering AST-based approach of Baxter et al. [10]. A linear-time efficient implementation using abstract syntax suffix trees is presented by Koschke et al. [14]. None of these approaches supports substitution of arbitrary subtrees.

Recent work by Roy and Cordy [15] is flexible in its level of abstraction, using pretty-printing as a fast hybrid between text-based, lexical and syntactical techniques. It also groups clones into clusters. The focus is on intentional modifications, and thus the goal is not to detect consistently renamed clones.

The anti-unification approach directly induces a suitable notion of distance between two statements, and the anti-unifier is a reference point which captures the common structure of the similar syntax trees. This makes the method general and at the same time flexible to the employment of varying similarity thresholds.

VI. Duplicate code detection tool

Our duplicate code detection tool is called Clone Digger. It is available under GNU General Public License and can be downloaded from the site `http://clonedigger.sourceforge.net`.

Clone Digger is written in Python and thus platform-independent. We use adapters which convert source files into an XML representation of their abstract syntax trees. Currently there are adapters for three languages: Python, Java 1.5 and Lua. Python abstract syntax trees are built using the standard CPython module "compiler". Java and Lua trees are built using ANTLR [16]. Adapters for other languages can be created, e.g. by using parser generators or using internal compiler representations.

Clone Digger can be run in two different ways: by invoking a Python script from the shell or by using the Eclipse plug-in which has been developed by Anatoly Zapadinsky during Google Summer of Code 2008. Clone Digger produces a HTML file with a list of clones. Each pair is reported statement by statement with color highlighting of differences. Examples are available on the tool webpage.

VII. Experimental results

We first present results of running Clone Digger on Python projects. Since we are not aware of other AST-based clone detection tools which support Python, we can only compare our tool with text-based or token-based tools independent of the programming language. We have chosen the tool DuDe [17] as comparison point. Table I shows results of running Clone Digger and DuDe on several open-source Python projects. We searched them for clones which occupy not less than 5 lines of code and have no more than 5 differences. As expected, the quality of clone candidates reported by the AST-based tool was better than the quality of token-based clones. For instance, some of the clones reported by DuDe occupied the end of one function and the beginning of the next function; such clones can't be refactored. If we split them, the size for one or both parts could be below the chosen threshold. Another expected observation is that DuDe is significantly faster than Clone Digger. Still, it is feasible to use Clone Digger as an online clone detection tool for small Python applications and as offline tool for larger ones.

Project	Size (loc)	Clones (%)		Time	
		CD	DuDe	CD	DuDe
Zope	21k	8.9	13.3	1m17s	7s
Plone	24k	12.27	15.73	2m21s	9s
NLTK	57k	10.29	15.87	9m37s	35s
BioPython	57k	11.83	10.10	31m57s	35s

TABLE I
Python test results

Next, we compare Clone Digger with the commercial AST-based detection tool CloneDR™ [10], as the Asta tool [5] which has similar structural detection to ours is not available for download. We used an evaluation version of CloneDR™ which reports only a subset (11 tuples) of detected clones. One can still check if CloneDR™ is able to detect a chosen clone by isolating it to a new file and re-running the tool.

We performed the comparison for Java as input language and ran CloneDR™ with default parameters on the source of the netbeans-javadoc project. The minimal clone size in the report was 6 code lines and there were maximum 5 differences. We next ran Clone Digger with the same thresholds. All clones reported by CloneDR™ were also found by Clone Digger, however, our tool detected significantly (50%) more: a total of 930 clones covering 3085 lines of code (Table II). Full reports are available at the Clone Digger site[1].

Project	Size (loc)	Clones (%)		Time	
		CD	CloneDR	CD	CloneDR
netbeans-javadoc	14K	21.48	14.82	7m57s	2m12s
eclipse-jdtcore	146K	14.24	18.09	2h43m	1h2m

TABLE II
Comparison to CloneDR™

[1] http://clonedigger.sf.net/IWSC09.zip

We have identified two types of valuable additional clones. First, CloneDR™ is only able to handle renamings, while Clone Digger handles replacements of subexpressions. Clone #3 in the report is an example. Second, Clone Digger supports parametrization of variable names and counts several equal renamings as one (appropriate for refactoring), thus resulting in smaller clone distances (e.g., clone #49).

Not all additional clones reported by Clone Digger are valuable. One reason is that CloneDR™ and Clone Digger have different meanings for the minimal size threshold. CloneDR™ supports clones made of several fragments, and one can specify a minimal desired total clone size, whereas Clone Digger does not support multi-fragment clones, and its size threshold refers to one piece. The measure of clone distance is also different. For instance, CloneDR™ considers the difference between *f(class1.variable1)* and *f(class2.variable2)* as 2, because it replaces *class1.variable1* with *class2.variable2* as a whole. However, Clone Digger considers class name and variable name separately, and measures the distance between these subtrees as 4.

Thus, to succeed in the first Java experiment, i.e., to find all clones which were detected by CloneDR™ required a low setting for the size threshold in Clone Digger. Predictably, this resulted in low precision: about 40% of the additional clones reported compared to CloneDR™ are false positives, especially among the small ones. The remaining 60% are real clones which CloneDR™ does not detect.

The goal of our second test, run on the project eclipse-jdtcore (ten times larger), was to minimize false positives. With the new thresholds, Clone Digger missed some clones which were detected by CloneDR™. However, about 25% of the clones reported by Clone Digger are not found by CloneDR™. Clone Digger built 10542 clusters for the 29661 statements in the code. The running times for the tools are still comparable, with CloneDR™ faster than Clone Digger.

The results show that Clone Digger can detect clones that CloneDR™ misses, at the price of running about three times slower. All tests were run with Python 2.4 on a PC with an AMD Athlon 5200+ processor and 4 GB of memory.

We have also categorized clones of different types for the netbeans-javadoc experiment. We selected the largest 50 clones: they included 6 exact copies, and 2 clones with replaced subtrees. The rest clones were clones with renaming. Most of the renaming clones were parametrized clones, i.e., there was more than one occurence of one replaced name. As previously explained, CloneDR™ doesn't handle parametrization; some of these clones were classified as such only because the numbers of references to such variables was small enough.

We next ran Clone Digger on the 4 Java projects in Bellon's survey [18], [19] to evaluate recall based on a reference corpus of clones. For the smaller projects, we used all references, for the large projects, a subset (5%). We omitted a few references which we did not deem true clones (e.g. unrelated setter and getter methods). Default parameters were a minimal clone size of 6 (as in the benchmark), and a maximal clone distance of 7 (decreased for smaller clones). Results are given in Table III.

Project	Size (loc, w/o cmt.)	Clones	References		Recall
			Found	Not	
netbeans-javadoc	8111	25.53%	23	23	50%
eclipse-ant	14671	10.53%	17	5	77.3%
eclipse-jdtcore	90009	31.32%	19	41	31.7%
javax-swing	95722	16.81%	19	20	48.7%

TABLE III
JAVA TEST RESULTS

Overall, the recall rates are comparable to the tools of the survey. In the limited time available, we have not performed a more detailed quantitative analysis. Overall, the limitations in recall are mainly due to clones with inserted statements (Clone Digger currently requires an exact match of statement sequences), and the handling of *object.method()* calls mentioned above. For the latter, Clone Digger should be changed to treat them as *method(object, ...)* which would expose a common top level and decrease anti-unification distance. For the former, step 3 of the algorithm could be changed to accept "imperfect" matches of statement sequences. In fact, these are already accepted if they are one level below in the AST, e.g., `if(...)` *block1* and `if(...)` *block2*. Another factor was that reference lengths included comments; without them, some short clones would fall under the stated threshold of 6 lines.

For eclipse-jdtcore and javax-swing, in spite of the low recall, Clone Digger identified most references as clones, but paired differently than in the benchmark: since clusters are built gradually, clones may end up in different clusters.

Precision is qualitatively very good. By limiting clone distance for smaller clones, we found that the vast majority of the reported pairs would be suitable candidates for refactoring.

In using Clone Digger on real-life large projects, we have found that automatically generated sources, tests and third party libraries should be excluded from consideration. First, the information on clones found in these files is useless for a user. Second, automatically generated sources and tests often contain large sequences of very similar statements. All these statements will be marked by the same cluster ID during the first phase and the amount of clone candidates found during the second phase will increase dramatically. Other practical aspects are discussed in [20].

VIII. CONCLUSIONS

We have presented a new definition of software clones based on anti-unification, which naturally captures syntactic structural similarity with high accuracy, and have developed an algorithm to detect clones based on this notion. We have implemented the algorithm in an open-source duplicate code detection tool named Clone Digger, which supports the Java, Python and Lua programming languages. Initial experimental results confirm that our method can identify clones which are missed by AST-based approaches which only support parameter renaming, although this comes at a cost in running time. We believe Clone Digger is effective and accurate enough to be used in the software development process.

This work was supported by INTAS grant 05-1000008-8144 and Romanian research grant 357/2007 *Continuous Quality Evaluation and Restructuring of Software*.

REFERENCES

[1] C. K. Roy, J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, Queen's University at Kingston, Ontario, Canada, 2007.
[2] R. Koschke. Frontiers of Software Clone Management. Frontiers of Software Maintenance track, Proc. 24^{th} IEEE International Conference on Software Maintenance, pp. 119-128, 2008.
[3] G. D. Plotkin. A note on inductive generalization. Machine Intelligence, vol. 5, pp. 153-163, Edinburgh University Press, 1970.
[4] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence, vol. 5, pp. 135-151, Edinburgh University Press, 1970.
[5] W. Evans, C. Fraser, F. Ma. Clone Detection via Structural Abstraction. Proc. of the 14^{th} Working Conference on Reverse Engineering, pp. 150-159, 2007.
[6] P. Bulychev, M. Minea. Duplicate code detection using anti-unification. Proc. Spring Young Researchers Colloquium on Software Engineering, Moscow, vol. 2, pp. 51-54, 2008.
[7] M.H. Sorensen, R. Gluck. An algorithm of generalization in positive supercompilation. Proc. International Logic Programming Symposium, pp. 465-479, MIT Press, 1995.
[8] C. Oancea, C. So, and S. M. Watt. Generalization in Maple. Proc. Maple Conference, pp. 377-382, Maplesoft, 2005.
[9] P. Bille. A Survey on Tree Edit Distance and Related Problems. Theoretical Computer Science, vol. 337(1-3), pp. 217-239, 2005.
[10] I. Baxter, A. Yahin, L.M. de Moura, M. Sant'Anna, L. Bier. Clone Detection Using Abstract Syntax Trees. Proc. 14^{th} IEEE International Conference on Software Maintenance, pp. 368-377, 1998.
[11] W. Yang. Identifying Syntactic Differences Between Two Programs. Software Practice and Experience, vol. 21(7), pp. 739-755, 1991.
[12] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.
[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, vol. 33(9), pp. 577-591, 2007.
[14] R. Koschke, R. Falke, P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. Proc. 13^{th} Working Conference on Reverse Engineering, pp. 253-262, 2006.
[15] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. Proc. 16^{th} IEEE International Conference on Program Comprehension, pp. 172-181, 2008
[16] T.J. Parr, R.W. Quong. ANTLR: A Predicated-LL(k) Parser Generator, Software Practice and Experience, vol. 25(7), pp. 789-810, 1995.
[17] R. Wettel, R. Marinescu. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments, Proc. 7^{th} International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2005.
[18] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis no. 1998, Universtität Stuttgart, 2002
[19] S. Bellon. Detection of Software Clones – Tool Comparison Experiment. `http://www.bauhaus-stuttgart.de/clones`, 2007
[20] P. Bulychev. Duplicate Code Detection Using Clone Digger. Python Magazine, September 2008.