2010 17th Working Conference on Reverse Engineering

Highly Configurable And Extensible Code Clone Detection

Benjamin Biegel and Stephan Diehl
University of Trier, Germany
Email: {biegel,diehl}@uni-trier.de

***Abstract*—Code clone detection is an enabling technology for plenty of applications, each having different requirements to a clone detector. In this paper we present a generic pipeline model of the code clone detection process. Based on this model we developed the JCCD code clone detection API for implementing custom clone detectors. By combining and parameterizing predefined API components as well as by adding new components, the pipeline model does not only facilitate to build new clone detectors, but also to parallelize the detection process.**

I. Introduction

A code clone is a code fragment in source code which is identical or similar to another code fragment—this description is likely to be the most popular text clone in code clone literature. Nevertheless, it is still an open issue to find a more accurate definition for this fundamental term [8], [15]. By now, over 40 approaches [13] have been introduced to detect clones—without having a consistent notion of *code clone*.

Code clone detection is an enabling technology for plenty of applications. It is used to find crosscutting concerns, to help understanding and improving source code, to reduce source code size, to expose malware, plagiarism and copyright infringement, to assist software evolution analyses, and to support refactoring detection [9], [13]. Each of these applications makes different demands on a clone detector.

In this paper we present a flexible and extensible approach to the code clone detection process. Due to the ambiguity of the term *code clone*, our main idea is to put the user in control. To this end we formally introduce a generic pipeline model. It exposes both the interplay as well as the distinction of all required steps in a clone detection process.

Based on this model, we developed the Java Code Clone Detection API (JCCD). It was designed to be flexible, extensible, accurate, and fast. We have released JCCD into open source (see http://jccd.sourceforge.net) under the new BSD license.

II. Generic Pipeline and its Implementation

In this section we will define every processing step of the generic pipeline which is presented in Figure 1. At the end, we will be able to give a procedural definition of the term *similarity pair*. Additionally, we will demonstrate how each step of this pipeline is implemented in JCCD.

A. Parsing

The pipeline gets plain source code as an input. The task in the parsing step is to make this source code suitable for further steps of the analysis. Note that we use the term *parsing* here in a more general sense. It does not necessarily require syntactical analysis.

Definition 1. *Let $\mathcal{F}$ be the set of all source code files, $\mathcal{U} = \mathcal{F}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathcal{A}$ be the set of all source units, $\mathcal{A}$ be the set of annotations, and $\wp(\mathcal{F})$ is the power set of $\mathcal{F}$. Then, the **parsing step** is defined by a function* $\mathsf{pars} : \wp(\mathcal{F}) \longrightarrow \wp(\mathcal{U})$.

Briefly, the parsing step transforms source code into source units. A **source unit** $(f, l_{start}, c_{start}, l_{end}, c_{end}, A) \in \mathcal{U}$ refers to a fragment of a source file f which starts at position c_{start} of line l_{start} and ends at position c_{end} of line l_{end}. A source unit might be a subtree of an AST, a line, a subgraph of a program dependence graph, or the like. The representation of a source unit depends on the used approach (e. g. text-based, token-based, AST-based, or metric-based). Furthermore, every approach requires different additional techniques like a line extractor, a lexer, or a parser.

In JCCD source units are represented by subtrees of an AST. First, such a tree reflects the syntactical structure of a document at a certain level of detail. Hence, we are able to take syntactical information into account, while irrelevant information like whitespaces or comments need not be represented in the AST. Second, it is less sensitive to code restructuring. Thus, it is less sensitive to minor code modifications. JCCD uses an automatically generated parser by ANTLR [11].

In Figure 1 we notice two branches which merge in the same processing step called *pooling*. This suggests that two different file sets can be included in the analysis. Only code clones between those sets will be detected. However, it is also possible to analyze only one file set. In other words, the lower branch is optional.

B. Preprocessing

The goal of the preprocessing is twofold: to normalize a set of source units and to add additional annotations. Normalization of a set of source units turns them into a regular form and thus makes different source units more similar.

Definition 2. *The **preprocessing step** is defined by a function* $\mathsf{prep} : \wp(\mathcal{U}) \longrightarrow \wp(\mathcal{U})$.

As input the preprocessing step gets a set of source units. It can change this set by modifying, removing or adding source units or annotations.

1095-1350/10 $26.00 © 2010 IEEE
DOI 10.1109/WCRE.2010.34

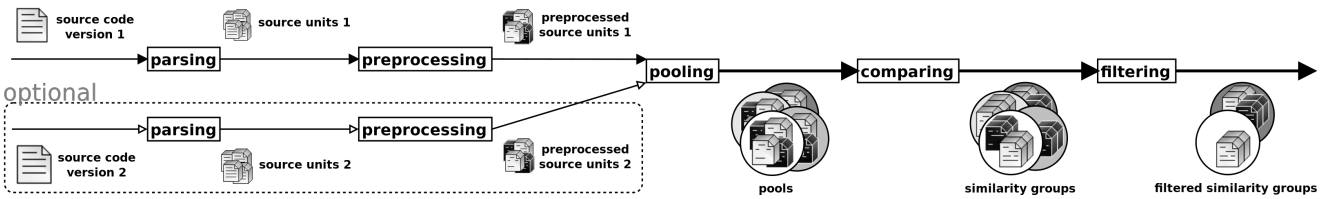

Fig. 1. Proposed Generic Code Clone Detection Pipeline.

In JCCD preprocessing is actually implemented by several cascaded preprocessors. Every preprocessor gets an (preprocessed) AST as input and returns a preprocessed AST as output. The user is free to select which preprocessors are to be used. Such a configuration is likely to have a significant impact on the recall and precision of the overall results.

A preprocessor is able to annotate, remove, collapse, and group AST-nodes. Some preprocessors normalize the AST with the above-mentioned operations, such as removing modifiers, generalizing variable names, or simplifying fully qualified identifiers.

Preprocessors can also compute new annotations based on the annotations set by previous preprocessors. These annotations can either be important for further preprocessors or subsequent phases of the pipeline. For example, annotations can enable to remove getter and setter methods, remove redundant parentheses, mark the scope of variables, or parameterize variable names consistent within a subtree.

C. Pooling

Next, preprocessed source units are grouped into different sets, called *pools*, based on user-defined criteria. Usually, these criteria are characteristics that can be directly read from the source unit and its annotations without comparing it to another one. For example, source units might be put into the same pool if they have the same variable name, the same numeric value, the same syntactical relations within the source code, or the same annotations (computed in the preprocessing). Thus, the pooling step enables a preselection of code clone candidates.

Definition 3. *The **pooling step** is defined by a function* $\mathsf{pool} : \wp(\mathcal{U}) \longrightarrow \wp(\wp(\mathcal{U}))$.

Thus, pooling yields a set $(P_1, \ldots, P_n)$ where each $P_i \subseteq \wp(\mathcal{U})$ is called a **pool**. Source units which are not in the same pool are not considered candidates for similarity pairs.

D. Comparing

The division of source units into pools allows the comparing step to apply a divide-and-conquer strategy. All of the given pools will be processed sequentially by comparing all contained source units recursively.

Definition 4. *The **comparing step** is defined by a function* $\mathsf{comp} : \wp(\wp(\mathcal{U})) \longrightarrow \wp(\wp(\mathcal{U}))$. *Let $\sim$ be a custom equivalence relation on $\mathcal{U}$. Then, we require that if* $(G_1, \ldots, G_n) = \mathsf{comp}(P_1, \ldots, P_k)$ *then* $\forall 1 \leq i \leq n : \exists 1 \leq j \leq k : \exists u \in P_j : G_i = \{x | u \sim x, x \in P_j\}$.

The comparison step further subdivides the pools into candidate sets G_i. More precisely, if $u \in G_i$ then $G_i = [u] \cap P_j$ where $[u]$ is the equivalence class of u. Or, in other words G_i contains all elements of P_j which are similar to u.

In JCCD the comparison is realized by using a user-defined set of comparators. Each comparator decides if two subtrees satisfy particular characteristics. Two subtrees are marked as similar only if the combination of all comparators is true. Actually, there are two different types of comparators: AND- and OR-comparators.

Definition 5. *Let S be the set of all subtrees in an AST. A function $c : S \times S \to \{true, false\}$ is called **comparator**.*

*Let C be a set of comparators. C is completely divided into the partitions $C_\wedge, C_\vee$ with $C = C_\wedge \dot\cup C_\vee$ where $C_\wedge$ contains all **and-comparators** and $C_\vee$ contains all **or-comparators**.*

To compare two subtrees $s_1, s_2 \in S$ the comparators are applied as follows:

$$\bigvee_{c \in C_\vee} c(s_1, s_2) \wedge \bigwedge_{c \in C_\wedge} c(s_1, s_2)$$

*Two subtrees are called **similarity pair** if the overall result of the selected comparators is* `true`.

E. Filtering

At the end, the filtering step is responsible for removing non-relevant candidate sets out of the result set. As before, the filter criteria are selected by the user.

Definition 6. *The **filtering step** is defined by a function* $\mathsf{filt} : \wp(\wp(\mathcal{U})) \longrightarrow \wp(\wp(\mathcal{U}))$. *We require that* $\mathsf{filt}(G) \subseteq G$.

For example, to get a more accurate result set in JCCD it is possible to select a filter for removing all similarity groups which are already represented by an enclosing group.

In addition, by using elementary set operations JCCD supports to tailor the result set. For example, in a first run the user could adjust the pipeline in the way that JCCD only detects code clones *useless* for the specific application (false positives). Next, the user could subtract these false positives of the code clones of a second differently adjusted run in order to get a more accurate result set.

F. Putting it all together

Figure 2 illustrates how the function gccd, i.e. the generic pipeline, combines all intermediate steps and results.

Definition 7. *Let the function* $\mathsf{gccd} := \mathsf{filt} \circ \mathsf{comp} \circ \mathsf{pool} \circ \mathsf{prep} \circ \mathsf{pars}$ *be the concatenation of all steps in our generic pipeline.*

$$\underset{\textit{files}}{\wp(\mathcal{F})} \xrightarrow{\mathsf{pars}} \underset{\textit{source units}}{\wp(\mathcal{U})} \xrightarrow{\mathsf{prep}} \underset{\substack{\textit{preprocessed}\\ \textit{source units}}}{\wp(\mathcal{U})} \xrightarrow{\mathsf{pool}} \underset{\textit{pool set}}{\wp(\wp(\mathcal{U}))} \xrightarrow{\mathsf{comp}} \underset{\substack{\textit{similarity}\\ \textit{groups}}}{\wp(\wp(\mathcal{U}))} \xrightarrow{\mathsf{filt}} \underset{\substack{\textit{filtered sim.}\\ \textit{groups}}}{\wp(\wp(\mathcal{U}))}$$

Fig. 2. The function gccd.

At this point, we have a generic pipeline enabling us to give a procedural definition of the term *similarity pair*:

Definition 8. *Let F be a set of source files. Then each pair $(u_1, u_2) \in \mathsf{gccd}(F)$ is called a **similarity pair**.*

We chose the term *similarity pair* and not the term *clone pair*, because one can choose functions for each of the phases, such that the computed pairs would not fit any common notion of a clone, but could still be useful for a certain application.

In JCCD the user is free to control the behavior of the pipeline by removing, replacing, or adding components. For the current version of JCCD we have implemented more than 30 comparators, over 40 preprocessors, and three different pooling strategies.

III. Evaluation

In this section we evaluate the suitability and the competitiveness of JCCD. To this end, we compare JCCD with the code clone detection tool CCFinder (10.2.7.1)[7]. Obviously, the fundamental difference of both implementations is that JCCD is AST-based and CCFinder is token-based. We still choose CCFinder because it is well-known, sophisticated, and has already been successfully applied in industry. Furthermore, the tool is designed for a fast analysis of large software projects. Actually, it has only linear computational complexity.

For our study, we have accomplished several test runs on 4 different open source projects. All the test runs were performed on the same workstation (*2.33GHz quad-core CPU, 16GB of memory, Ubuntu Linux 9.04 64bit*). Table I gives an overview of the used configuration.

TABLE I
Configuration of CCFinder and JCCD.

Properties of a clone	**CCFinder**	**JCCD**
minimum 12 token sets	x	x
minimum 30 tokens	x	x
is a syntactical unit (AST-subtree)		f
parameterized identifiers	x	x
parameterized string, char, numerical, and boolean literals	x	x
subsume overlapping clones	f	x
convert to compound block	f	x
concatenation of tokens	f	n
Remove/ ignore tokens		
assertion, package, import, modifiers	f	x
redundant parentheses in return statements	f	x
array initialization tables	f	x
simple delegations, getter, setter methods	f	x
empty methods, interfaces	f	x
repeated code	f	n

x = selected f = fixed n = not supported

For the evaluation we have chosen a configuration which is as similar to CCFinder as possible. For that, we have had to adjust several components of JCCD. However, some details were not considered in this first evaluation (like concatenation of tokens, removal of repeated code, or other uncovered functions of CCFinder).

By comparing the adjustable parameters we determine that many parameters of CCFinder are fixed. To change these parameters the user has to reprogram parts of the preprocessors which are available as python scripts. In contrast, JCCD is very flexible. Most parameters are adjustable. Beyond, whole parts of the detection pipeline can be exchanged.

A. Evaluation of the Clone Pair Coverage

The intention of the following subsection is to determine the coverage of the clone pair sets which were detected by CCFinder and JCCD. For this preliminary evaluation the clone pair sets were taken from the source code of jEdit (4.3.1)[5]. Overall CCFinder found 4489 clone pairs and JCCD 1369. This spread is based on the different approaches. CCFinder is able to find clone pairs which enclose any sequence in the source code. In the current version of JCCD only subtrees of an AST can form a clone pair. In other words, neighboring subtrees (sequences) are not combined to clone pairs.

In Table II the coverage of the clone pairs is presented. We selected a simple algorithm to compute the coverage of two clone pairs.

Definition 9. *Let $lines(s) = \{(f,l) | s = (f, l_s, c_s, l_e, c_e, a) \wedge l_s \leq l \leq l_e\}$ denote the lines contained within a source unit s. The **coverage** of a similarity pair p by a set of similarity pairs P is defined as $coverage(p, P) = max\{cov(p, p') | p' \in P\}$ where*

$$cov((s_1, s_2), (s_1', s_2')) = \begin{cases} 0 & \text{if } s_1 \sqcap s_1' = 0 \\ & \quad \vee s_2 \sqcap s_2' = 0 \\ \frac{s_1 \sqcap s_1' + s_2 \sqcap s_2'}{2} & \text{otherwise} \end{cases}$$

and $s \sqcap s' = \frac{2 * |lines(s) \cap lines(s')|}{|lines(s)| + |lines(s')|}$.

As can be seen in Table II, the results are divided into four groups. The column *large* includes the ratio of all clone pairs which overlap *at least* 80%. Not included are fully overlapping clone pairs which can be found in the column *full*. For example, if $\mathbf{J}$ is the set of clones found by JCCD and $\mathbf{C}$ is the set of clones found by CCFinder, then

$$\frac{|\{p \in \mathbf{J} | 0.8 \leq coverage(p, \mathbf{C}) < 1\}|}{|\mathbf{J}|} = 53,07\%.$$

Analogously, the column *small* includes the ratio of all clone pairs which overlap *at most* 80%. Not included are non-overlapping clone pairs which can be found in the column *none*. The first line relates to clone pairs of CCFinder which overlap clone pairs of JCCD. Analogously, the second line relates to clone pairs of JCCD which overlap clone pairs of CCFinder.

As shown in previous studies CCFinder has a good relation of recall and precision [12]. Therefore, the results of this preliminary evaluation suggest that JCCD has a low recall and a high precision. Despite the different approaches we determine that nearly 20% of the CCFinder clone pairs are

TABLE II
COVERAGE OF CCFINDER AND JCCD.

	full	large	small	none
CCFinder ⊂ JCCD	3.37%	16.06%	6.59%	73.98%
JCCD ⊂ CCFinder	11.62%	53.07%	30.19%	5.12%

detected by JCCD with an overlapping of at least 80%. Furthermore, nearly 65% of all JCCD clone pairs are confirmed by CCFinder. By comparing the results of both implementations we assume that the quality of the result set of JCCD is high. In particular only a small part of 5.12% of the JCCD clone pairs is not covered by CCFinder.

B. Qualitative Evaluation

In this section we take a closer look to clone pairs which were only detected partly or exclusively by one single approach. In both sets the ratios of full coverage are relatively small because both implementations use different marking boundaries of the clone pairs. JCCD summarizes the exact boundaries of syntactic units, while CCFinder uses *leading tokens* (like ';', '{' or ':') to enclose a code fragment. For example by marking a whole method JCCD starts by the first token of this sequence (like the method type) and CCFinder starts by a leading token which appears immediately before the method (like a closing curly bracket of the previous method). This has the consequence that some JCCD clone pairs have a small overlap with CCFinder but describe exactly the same code fragments. This occurs, in particular, in code fragments with a large ratio of comments or empty lines. Since most methods in jEdit are described by an introductory comment block, this has a strong influence on the coverage values.

As might be expected, the main reason for the low coverage of CCFinder clone pairs in JCCD is the use of different approaches. CCFinder is able to detect token sequences which might enclose a set of neighboring syntactical units. The `if`-constructs are such syntactical units and JCCD is able to detect both as a clone pair when the configuration has been adjusted accordingly (smaller token length). In this case, the lengths of both constructs are too small. However, CCFinder marks also a statement (last line) which represents a further adjacent syntactical unit. During this evaluation we have not found any clone pair of CCFinder which had no overlap due to another reason.

By analyzing clone pairs which are not detected by CCFinder we determine different reasons which are listed below:

- **View of Token Length:** CCFinder subsumes some token sequences and counts a whole sequence as one single token. This process reduces the length of a token sequence. Thus, a lot of clone pair candidates are ignored in the analysis of CCFinder which are still detected by JCCD. We checked several clone pairs of this kind and confirmed that JCCD has detected them correctly. Some other uncovered subsumptions of CCFinder might also lead to a spread in the result set of both implementations.
- **Generalizing of Identifiers:** In particular identifiers will be subsumed by CCFinder to a single token, modifiers will be ignored and identifiers will be parameterized.
- **Boolean Literals:** JCCD detects identical code fragments that differ only in `boolean` literals. According to the documentation CCFinder is able to find these pairs, too. We could not determine why this kind of clone pairs was not detected.

Briefly, besides a few unexplained inconsistencies we determine that the length of most of the non-overlapping clone pairs is too small (under 30 tokens) in the view of CCFinder. The 30% of the JCCD clone pairs which have only a small coverage (less than 80%) are in all considered cases completely enclosed by a bigger clone pair of CCFinder. This phenomenon occurs most frequently in nested constructs (like `if...else if...`etc.). CCFinder is able to mark any sequence of such a construct as clone pair. In the AST representation the second `if`-construct is a descendant and not a sibling of the first one. In the current version of JCCD it is only possible to select whole subtrees as clone pairs.

TABLE III
RUNTIMES OF CCFINDER AND JCCD.

	jEdit	**Tomcat**	**Vuze**	**Android**
NLOC	107,278	167,938	477,615	1,397,114
Files	531	1,143	3,234	9,937
JCCD Runtimes				
parsing	7.15s	10.07s	21.41s	65.87s
preprocessing	4.52s	6.72s	12.00s	35.62s
pooling	0.27s	0.38s	0.86s	7.64s
comparing				
non-parallelized	5.53s	16.97s	127.36s	1,385.72s
parallelized	3.10s	7.24s	65.48s	536.31s
speedup factor	*1.784*	*2,344*	*1.945*	*2.584*
filtering	0.00s	0.02s	0.04s	0.10s
Total Runtimes				
JCCD				
non-parallelized comparing	17.48s	34.17s	161.67s	1,494.96s
parallelized comparing	15.03s	24.29s	99.79s	645.59s
speedup factor	*1.163*	*1.407*	*1.620*	*2.284*
CCFinder	22.67s	51.45s	108.16s	746.04s

C. Quantitative Evaluation

In this section we present the results of several performance tests by running both implementations over 4 different open source projects of varying size and application domains. We have selected the following projects: jEdit(4.3.1)[5], Apache Tomcat (6.0.24)[2], Vuze (4.3.1.4)[14], and Android (1.5 cupcake)[1]. The results of the performance tests with JCCD and CCFinder are reported in Table III.

1) JCCD Runtimes: By comparing the performance results of each processing step we determine that the comparing step is the bottleneck of the whole analysis. This fact leads us to improve this step by exploiting the multi-core architecture of the workstation used for the tests.

The above version of JCCD processes all pools one after another. By splitting this task into multiple threads a multi-core system is able to process a set of pools simultaneously.

We have adapted the comparing step to this functionality and repeated the test runs. Actually, only the body of a loop in the comparing step had to be moved into a separate thread and a data structure of type `HashMap` had to be turned into type `ConcurrentHashmap`.

The parallelization has not only a significant impact on the runtime of the comparing step, but also on the total runtime. Almost all comparing runs have a speedup about the factor 2. In particular, the complete analysis of Android takes half the time of the single-threaded version.

2) Comparison with CCFinder: Next, by comparing the runtimes with CCFinder we demonstrate that they are acceptable in practice. Overall, we note that the runtimes of JCCD are competitive to the runtimes of CCFinder. Only by analyzing the biggest project Android the single-threaded JCCD takes twice the time of CCFinder. However, the multi-threaded version is competitive with a speedup factor of 1.156.

D. Threats to Validity

Construct validity: We are not the first to face the problem of comparing the performance of code clone detectors. As these tools typically find different kinds of clones and only to a certain extent the same clones, the pure performance numbers can only be compared with caution — even if some kind of coverage and quality analysis was combined with the performance analysis as done in this paper.

Internal validity: While it seems reasonable to assume that the speedup when using multiple cores was mainly due to the additional processing power, other effects like caching, changed garbage collection behavior and the like may have influenced the results.

External validity: To allow for some generalizability of our findings, we performed our comparison with four open source projects of varying size and application domains, but all written in Java. We cannot claim that our findings can be held true for other programming languages, other code clone detectors, or software projects with larger size.

IV. Related work

Current clone detection approaches can be roughly categorized by the kinds of information they process: strings, tokens, trees, program dependence graphs, metrics, or hybrids [13].

Baxter et al. introduce a pioneering AST-based clone detection approach [3] which is representative for most tree matching techniques. A parser transforms source code into an AST. Optionally, its subtrees are partitioned into buckets based on a hash function. Only subtrees within the same bucket are compared to each other by fuzzy tree matching. In contrast, in our generic pipeline model all phases are clearly distinguishable. This enables to build custom clone detectors and to substitute particular phases of the pipeline.

CloneDetective implements a pipelined approach for extensible token-based clone detection [6]. In their approach the actual detector is a single component. In contrast, JCCD is AST-based and the detection process is further subdivided into phases.

Our generic pipeline is similar to the one proposed by Roy et al. [12]. While they use the pipeline to classify and compare different clone detection techniques, we used our pipeline model to receive a flexible architecture. Moreover, in JCCD their ”Match Detection” phase is more refined. Nevertheless, their pipeline-based description of various techniques supports our claim that many of these techniques could be implemented with JCCD.

JCCD was already presented in a tool demonstration [4]. It was originally developed to provide a customizable code clone detector for our refactoring identification framework [16]. Recently, JCCD has also been used to enable the detection of more complex refactorings [10]. It computes a candidate set based on code similarity and applies additional heuristics to further reduce this set. The implementation consists of extending several phases of the JCCD code clone detection pipeline. The integration enables refactoring-specific optimizations at early stages of the pipeline.

V. Conclusions

In this paper we formally introduced a generic pipeline model which allows a flexible and extensible approach to the code clone detection process. Further, we presented JCCD, a versatile API for implementing custom clone detectors based on the generic pipeline. Despite the overhead caused by the modularization our evaluation shows that JCCD has a comparable performance to a well-established clone detector, and that it has a good precision. Moreover, we exploited parallel processing on a multi-core computer to achieve better runtimes by simply extending one particular phase of the pipeline.

References

[1] Android, http://source.android.com/, Feb. 2010.
[2] Apache Tomcat, http://tomcat.apache.org/, Feb. 2010.
[3] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *ICSM*, 1998.
[4] B. Biegel and S. Diehl, “JCCD: A flexible and extensible API for implementing custom code clone detectors,” in *ASE*, 2010.
[5] jEdit, http://www.jedit.org/, Feb. 2010.
[6] E. Jürgens, F. Deissenboeck, and B. Hummel, “CloneDetective - a workbench for clone detection research,” in *ICSE*, 2009.
[7] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” *IEEE TSE*, 2002.
[8] C. Kapser, P. Anderson, M. W. Godfrey, R. Koschke, M. Rieger, F. V. Rysselberghe, and P. Weißgerber, “Subjectivity in clone judgment: Can we ever agree?” in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, 2007.
[9] R. Koschke, “Frontiers on software clone management,” in *ICSM*, 2008.
[10] D. Neu, “AST-basierte Erkennung von komplexen Refactorings,” Diploma Thesis (in German), University of Trier, Germany, 2009.
[11] T. J. Parr and R. W. Quong, “ANTLR: A predicated- *LL(k)* parser generator,” *Software, Practice and Experience*, 1995.
[12] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, 2009.
[13] C. Roy and J. Cordy, “A survey on software clone detection research,” Queen’s University at Kingston, Canada, Tech. Rep., 2007.
[14] Vuze, http://vuze.sourceforge.net/, Feb. 2010.
[15] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, “Problems creating task-relevant clone detection reference data,” in *WCRE*, 2003.
[16] P. Weißgerber and S. Diehl, “Identifying refactorings from source-code changes,” in *ASE*, 2006.