CloneDetective – A Workbench for Clone Detection Research

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{juergens,deissenb,hummelb}@in.tum.de

Abstract

The area of clone detection has considerably evolved over the last decade, leading to approaches with better results, but at the same time using more elaborate algorithms and tool chains. In our opinion a level has been reached, where the initial investment required to setup a clone detection tool chain and the code infrastructure required for experimenting with new heuristics and algorithms seriously hampers the exploration of novel solutions or specific case studies. As a solution, this paper presents CloneDetective*, an open source framework and tool chain for clone detection, which is especially geared towards configurability and extendability and thus supports the preparation and conduction of clone detection research.*

1. Customizable Clone-Detection

Software clones increase maintenance efforts [15] and have negative effects on program correctness [7]. Consequently, significant research effort has been dedicated to tools that detect cloned code in large software systems [10, 15]. Such clone detection tools have been applied to investigate various code-cloning related research questions, including its causes and evolution and its effects on correctness and maintenance efforts. Furthermore, clone detectors have been applied to a large variety of tasks in both research and practice, including quality assessment, software maintenance and reengineering, identification of crosscutting concerns, plagiarism detection and investigation of copyright infringement [10, 15].

Each of these tasks imposes different requirements on the clone detection process and its results [18]. For example, the clones relevant for redundancy reduction, *i. e.*, clones that can actually be removed, differ significantly from the clones relevant for plagiarism detection. Similarly, a clone detection process used at development time, *e. g.*, integrated in an IDE, has different performance requirements than a detection integrated in a nightly build. Moreover, even for a specific task, clone detection tools need a fair amount of tailoring to adapt them to the peculiarities of the analyzed projects. Simple examples are the exclusion of generated code or the filtering of detection results to retain only clones that cross project boundaries. More sophisticated, one may want to add a pre-processing phase that sorts methods in source code to eliminate differences caused by method order or to add a recommender system that analyzes detection results to support developers in removing clones.

Problem. These examples illustrate that the required customization does not only affect the clone detection algorithm itself. Instead, it concerns all phases of the clone detection process, namely input, pre-processing, detection, post-processing, and output. In many cases, clone detection researchers are not concerned with all of these phases but concentrate on one or two individual phases. To avoid rebuilding the detection phases they are not primarily interested in, they require a clone detection tool with an open, flexible architecture that allows to fine-tune individual processing phases and, importantly, to add novel processing steps or disable existing ones. However, most of the clone detection tools used in research and practice are either unpublished research prototypes [6, 12] or closed source [1, 8, 11, 16, 17]. The few remaining open source clone detectors [2, 5, 13] have not been designed for extensibility. This mostly reduces their application by others to black-box usage that does not well cater for the mentioned types of modification required for efficient task-specific tailoring. This lack of tailorability forces researchers to create new tools, which is costly and not always impossible.

Contribution. To alleviate this, we present CloneDetective, a workbench for clone detection research with extensive tailoring capabilities. CloneDetective treats the distinct phases of the clone detection process as first class citizens and, hence, allows to flexibly configure the detection process using a declarative configuration mechanism that does not require actual programming. Furthermore, CloneDetective supports the addition of supplementary processing steps in all stages of the clone detection process. Users of CloneDetective also benefit from it being open source as

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

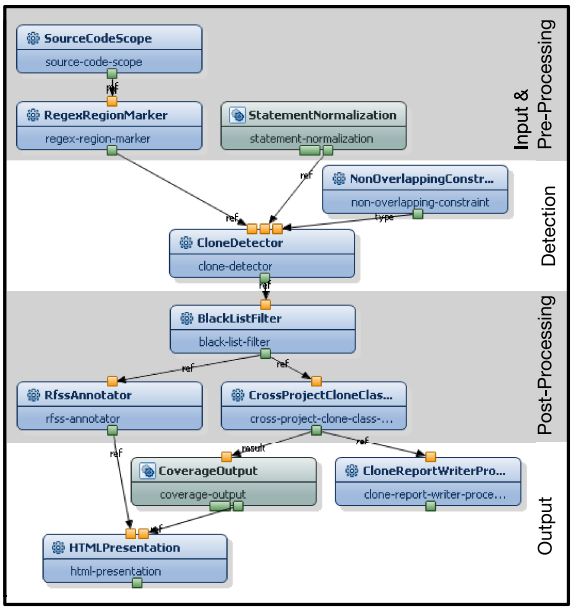

Figure 1. Clone detection configuration

they can freely alter existing processing steps or use them as inspiration for novel developments[1]. CloneDetective has been used successfully as a research platform and is also applied in several commercial projects [3, 4, 7].

2. Configurability and Extendability

The clone detection pipeline quite naturally can be split into several phases. In the *input and pre-processing phase*, code is read and the required program representation created. During *detection*, similar code fragments are identified. Results are then *post-processed* and presented to the user in various formats in the *output* phase. Each of these phases can be further subdivided, leading to more fine-grained and complex pipelines. In a token-based clone detector, *input* can, *e. g.*, be split into *file input*, *tokenization*, *normalization* and *filtering*.

As a single fixed pipeline is unnecessarily restrictive, *CloneDetective* is built upon the *ConQAT* infrastructure [4]. This allows the clone detection pipeline to be specified and parametrized using ConQAT's data-flow configuration language, where each processing unit (called a *processor*) is described by a single Java class. The core of the CloneDetective basically provides a rich set of processors for ConQAT. So besides reusing existing configurations for clone detection, the user may build her own configuration using ConQAT's graphical configuration tool, or even provide new building blocks for a detection by writing Java code.

An example of a (slightly simplified) clone detection configuration is given in Fig. 1, which is an annotated screen shot from the configuration tool. Initially the source code is loaded from disk and generated code is excluded. Additionally the (lazy) normalization pipeline is prepared and passed to the clone detector. This example demonstrates, that both data (source code) and algorithms (the normalization pipeline) can be passed to other processors. The *StatementNormalization* also is an example for a so called *block*, which is ConQAT terminology for a reusable sub configuration, *i. e.*, the StatementNormalization internally is not written in Java, but consists of further processors and blocks. The detection phase mainly consists of a single processor performing the core detection algorithm, which might be parametrized using additional constraints. The detection result is then filtered and the *redundancy free source statements (RFSS)* are calculated, before in the output phase clone coverage reports and statistics are generated as HTML pages and an XML clone report is written as input for further tools in the tool chain. Several settings, such as the minimal length of detected clones, are given as parameters to the processors, but are not shown in the screen shot. The options CloneDetective provides for each of these phases are detailed further in the next section.

3. Code Clone Detection

We give an overview of the the functionality CloneDetective offers for code clone detection along the detection phases and report on experiences from their application.

Input. The input components transform source files into a sequence of normalized program units. CloneDetective offers language-independent line- or word-based input processors. Basic normalization can be specified using regular expressions. For deeper control, tokenizers for multiple languages, including Java, C#, C/C++, Visual Basic, Cobol and PL/I are available. Their token stream can be normalized in order to ignore differences in comments, literal values or identifier names during detection. Normalization can be performed in a context-sensitive fashion, allowing, *e. g.*, for more conservative normalization of stereotype code regions, such as sequences of getters and setters. *Shapers* allow limitation of detected clones to boundaries of classes, methods or basic blocks. Since clones in generated code are uninteresting for most clone detection tasks, generated code can be filtered based on file names or file content. Notably, file fragments can be excluded as well, such as specific methods inside otherwise hand-written code.

Such tailoring flexibility turned out to be crucial to achieve recall and precision values required for application during continuous quality control at our industrial partners. For example, precision values of 90% could only be reached by filtering code from multiple generators and performing context sensitive normalization. In addition, this input processing extensibility also supports easy realization of ap-

[1] http://www.clonedetective.org/

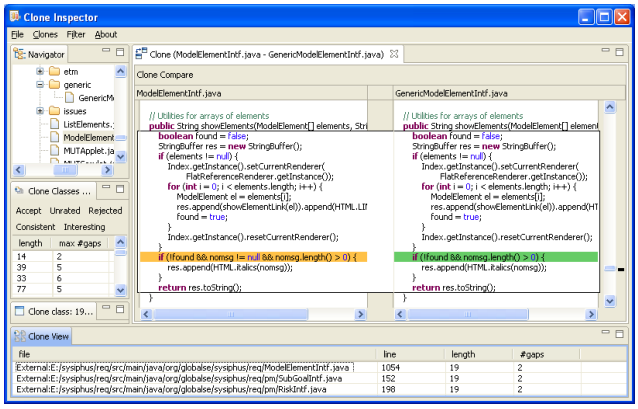

Figure 2. Integration into Eclipse

proaches suggested by others such as the use of region information for precision improvement [9].

Detection. The detection phase searches for similar subsequences in the program unit sequence created by the input phase. Currently, two different algorithms are available. Both work on an optimized suffix-tree created from the program unit sequence. *Ungapped* clone detection searches for identical sub-sequences, thus detecting code fragments that only differ (depending on normalization settings) in whitespace, identifier names, comments or literal values. Detection is simply performed by suffix tree traversal and runs in linear time and space. The novel *Gapped* clone detection additionally detects clones with statement modifications, additions or removals. The performance of these algorithms depends on the analyzed systems and overall detection configuration. Both are capable of analyzing systems of several MLOC which is sufficient for most industrial systems. The ungapped detection processes 10 MLOC in about 10 minutes on a 2.4 GHz CPU (without I/O time).

Post-processing. Post-processing steps work on clone result data to perform, amongst others, filtering and metric computation. Cross-project clone filters can be used to identify code cloned between different projects. Clone content filters can, *e. g.*, be used to identify clones containing issue numbers in their duplicated code. Furthermore, clone blacklisting can be used to exclude individual clones.

Besides filtering, several metrics can be computed. *Clone coverage* expresses the ratio of the system affected by duplication and thus estimates the probability that a change to an arbitrary program statements needs to be performed in more than one place. *Number of Statement Occurrences* expresses the average number of times a statement is cloned and thus estimates the number of statements that need to be touched if an arbitrary statement is modified. *Redundancy Free Source Statements* computes the system's size if redundancy would be removed perfectly, taking overlapping clones into account. The cloning relationships can also be represented as a graph. ConQAT's graph processing capabilities can then be used on the clone graph, *e. g.*, for visualization, clustering or to compute centrality metrics.

We repeatedly relied on the filtering capabilities when applying clone detection during quality assessments and maintenance support. Cross-project clone filters allowed us to inspect inter-product cloning in an industrial software product line and thus helped to discover weaknesses in the applied reuse process. During analysis of a Cobol system, filters that remove clone classes whose clones are not in the same file were beneficial to identify candidates for easy removal, since Cobol provides a simpler mechanism to call a procedure in the same file, than from another file. Clone fingerprints, that provide identification of a clone class across different reports and are robust against changes to non-cloned code, allow developers to permanently exclude false positives that have escaped tailoring efforts. Furthermore, the extensible nature of ConQAT allowed us to correlate clone metrics with data produced by other static analyses to provide a comprehensive quality dashboard to developers.

Output. Output processors prepare clone result data for use by tools or users. Depending on the use case, different options are available. An XML clone report can be written for use by tools. It's format is extensible to allow new processors to store additional information. For easy usability or integration into a project dashboard, clone results can be rendered to HTML. Clone lists give access to detailed information. Tree maps provide a system-wide overview of cloning. Furthermore, clone data can be historized and trend charts can be computed, depicting the development of cloning metrics over time.

Ecosystem. A stand-alone report viewer and IDE integration for Eclipse and Visual Studio.NET are available to support working with clone reports. In combination with the output processors, the provided visualizations support clone inspection on various levels of abstraction, ranging from the code-level compare view to system level cloning tree maps. Both the stand-alone viewer and the Eclipse plug-in implement syntax-highlighting for all supported languages. They provide a dedicated compare-view that shows cloned code fragments side-by-side, highlighting inconsistencies, as depicted in Figure 2. A *SeeSoft* view displays clones in a bars-and-stripes fashion, providing less detail, but better overview. Numerous filters are available to manage large clone reports without having to re-execute detection. A plug-in for Visual Studio.NET supports clone detection from inside the IDE and notifies developers when they are changing cloned code. Both plug-ins support fingerprint-based clone blacklisting.

The compare-view and SeeSoft view offered by the stand-alone viewer substantially increase clone inspection

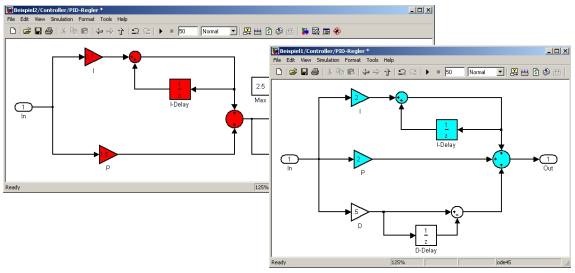

Figure 3. Visualization of clones in Matlab

productivity. From our experience, simple text-based reports are simply too unusable for large scale software systems. Hence, such tool support is essential to perform empirical studies as presented in [7].

4. Beyond Code

Besides code clone detection, CloneDetective implements the first clone detection algorithm for graph-based modeling languages [3]. More specifically, it finds similar sub-models in Matlab/Simulink files, which are used for describing control theoretic algorithms in a data-flow fashion. Such models are commonly used in the automotive domain. An example of such a clone and its visualization within Matlab is shown in Fig. 3. Analogously to code clone detection, the phases of input, pre-processing and normalization, detection, post-processing, filtering and output are supported by a family of reusable processors that ease the development of clone detectors for other graph-based modeling languages. CloneDetective can thus serve as a workbench for both code and model based clone detection research and tool development. Parts of the framework have already been used as a basis for improved model clone detectors by other researchers [14].

5. Conclusion

This paper introduced *CloneDetective*, an open source framework and tool chain for clone detection. Due to its configurability and extendability we consider it an ideal platform for research and experimentation in this area, which is backed up by the results of our group. At the same time it is more than just a "research toy", as the algorithms and their implementation scale well to serious amounts of code as experienced in various industrial cooperations. For both academic and commercial users of our tool chain, detection is only the first step. Often more important than the bare detection results and statistics is a thorough inspection and evaluation of the results, possibly followed by tailoring the detection parameters and algorithms to yield more precise results. All of these steps are supported by individual parts of our tool chain.

While in our opinion *CloneDetective* implements the current state of the art in token-based clone detection, we are striving to continuously improve and extend it. Future work of our group will be based upon and extends this tool chain. We hope that our tool further stimulates clone detection research and invite researchers and practitioners to both utilize and contribute to *CloneDetective*.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, Washington, DC, USA, 1998. IEEE.

[2] Copy paste detector. http://pmd.sourceforge.net/cpd.html.

[3] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. of ICSE '08*. ACM, 2008.

[4] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM '99*. IEEE, 1999.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE '07*, 2007.

[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of ICSE '09*. IEEE, 2009. To appear.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(0157):654–670, 2002.

[9] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *Proc. of IWPSE '04*. IEEE, 2004.

[10] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[11] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. WCRE '06*. IEEE, 2006.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[13] M. M. Peter Bulychev. Duplicate code detection using anti-unification. Proc. of SYRCoSE, 2008.

[14] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proc. of ICSE '09*. IEEE, 2009. To appear.

[15] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University at Kingston Ontario, Canada, 2007.

[16] Simian. http://www.redhillconsulting.com.au/products/simian/.

[17] Simscan. http://blue-edge.bg/simscan.

[18] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proc. of WCRE '03*. IEEE, 2003.