CloneWorks Reference Manual

Jeffrey Svajlenko
jeff.svajlenko@gmail.com

Chanchal K. Roy
chanchal.roy@usask.ca

November 13, 2016

Contents

1 CloneWorks

CloneWorks is a fast and flexible near-miss clone detector for large-scale clone detection experiments. It gives the user full control over the source normalizations, transformations, filtering and processing before clone detection, for general and targetted clone detection experiments. CloneWorks is very fast, executing for 250 million lines of code (LOC) in just four hours in an average workstation. Its input partitioning strategy allows it to handle any input size within the memory constraints of an average workstation.

Full details on CloneWork's algorithms and architecture can be found in its publication[1]. A link to the paper is provided at `http://www.jeff.svajlenko.com/cloneworks`. There is also a demo video available.

2 Contact

Please send us your questions, suggestions, feedback and bug reports. Our contact details are available on the front page of this document. We are happy to assist you in using CloneWorks.

3 Setup

CloneWorks can be used on Linux and other Unix-like systems. It has been tested on Ubuntu and OS X. For use on Windows, consider the use of "Bash on Ubuntu on Windows", a new Windows 10 feature.

CloneWorks is available for download at `http://www.jeff.svajlenko.com/cloneworks`. Download the latest version and extract it on your machine.

CloneWorks requires Java-1.7 or later. Java can be installed using your package manager (OpenJRE/OpenJDK), or downloaded and installed from Oracle (`http://www.oracle.com/`).

CloneWorks requires FreeTXL 10.6 or later, which can be downloaded from `http://www.txl.ca`. The TXL scripts need to be compiled for your computer. Enter the txl/ directory and run make from the command-line.

CloneWorks requires configuration for your available memory. By default, the input builder is set to use up to 2GB of memory, and the clone detector up to 4GB of memory. These can be modified by editing the cwbuild and cwdetect scripts. The input builder is probably fine with 2GB, unless you receive an out of memory error from the JVM. The clone detector should be set for your available memory, leaving room for system usage.

By defualt, CloneWorks must be executed from its installation directory. To run CloneWorks from another directory, the cwbuild and cwdetect scripts must be modified with `'INSTALL\_DIR=...'` set to the install directory (absolute path). If the installation directory is specified, cwbuild and cwdetect can also be added to the PATH variable for execution anywhere.

While setup is easy, it can be avoided by downloading CloneWorks as a virtual machine image (from `http://www.jeff.svajlenko.com/cloneworks`. This image was created with VMWare, and runs on the latest VMWare Player, which is free to download. The image is in an open standard, so it may work with other virtual machine software, such as VirtualBox.

4 Testing Setup

Once installed, CloneWorks can be tested by executing the following commands in the installation directory:

```
>./cwbuild  -i example/JHotDraw54b1/ -f JHotDraw.files \
            -b JHotDraw.fragments -l java -g function -c type3_conservative

>./cwdetect -i JHotDraw.fragments -o JHotDraw.clones -s 0.7 -mil 10

>./cwformat -f JHotDraw.files -c JHotDraw.clones -o JHotDraw.clones.csv -t csv
```

[1]The paper is currently in-progress. It will be posted upon successful publication.

Figure 1: cwbuild command-line usage

```
usage: cwbuild -i <path> -f <path> -b <path> -l <str [str2 st3 ...]> -g <str> -c <config>
               [-t <num>] [-mil <num>] [-mal <num>] [-mit <num>] [-mat <num>]
CloneWorks-InputBuilder - CloneWorks input builder.
-i,--input <path>                    Input subject system.  Either the root directory of
                                     the system, or a file containing a list of source file paths.
-f,--fileids <path>                  File to write the FileID<->FilePath mapping to.
-b,--blocks <path>                   File to write the parsed blocks to.
-l,--language <str [str2 st3 ...]>   The language of the input system.
                                     One of: {java,c,cpp,cs,python}.
-g,--granularity <str>               The block granularity.
                                     One of: {block,function,file}.
-c,--configuration <config>          The configuration file.  Name of a file in
                                     INSTALL_DIR/config/.
-t,--num-threads <num>               The number of threads to use per execution task.
                                     Default is number of available cores.
-mil,--min-lines <num>               Minimum code fragment size in (original)
                                     source lines.
-mal,--max-lines <num>               Maximum code fragment size in (original)
                                     source lines.
-mit,--min-tokens <num>              Minimum code fragment size in (original,
                                     pre-processing) parsed tokens.
-mat,--max-tokens <num>              Maximum code fragment size in (original,
                                     pre-processing) parsed tokens.
```

This performs Type-3 clone detection on JHotDraw. The clones are in the file example/JHotDraw.clones, the code fragments are in example/JHotDraw.fragments, and the mapping of ID to source-file paths are in example/JHotDraw.files.

5 Basic Usage

In this section we describe the basic usage of CloneWorks. This shows how to perform clone detection using the standard configurations. For advanced usage of CloneWorks, including providing your own code-fragment processor, also refer to Section 6.

The input builder (cwbuild) is used to prepare the source code for clone detection, which is performed by the clone detector (cwdetect)

5.1 Input Building (cwbuild)

The input builder is used to prepare the code fragments for clone detection. Here we show how to use the input builder with one of the included configuration files. The interface for the input builder is shown in Figure 1.

You must specify the directory of source code to process, a file to output the code fragments to, and a file that maps between the source file IDs and their source paths. This mapping is used to compact these output file sizes, in particular when a large number of files are processed. The language(s) of the source files of interest must be specified, as well as the code granularity for extraction.

The configuration of the input builder's source processing must also be provided. Configuration files are stored in the config/ directory, and referred to by their file name. The configuration is a list of the source transformations/normalization/filtering to apply to the code fragments, the granularity for term splitting, and the list of term processors to apply. These allow the user to completely customize the representation of the code fragments for clone detection. We discuss these further in Section 6 ('Advanced Usage'). In the following subsection we overview the existing configurations provided.

Optionally, the minimum and maximum size of the code fragments to consider, by original source line and pre-processed terms (after term splitting, before term processors). This can also be done with the clone detector, or can be specified here to reduce the overall size of the code fragments file.

The target number of threads per task can also be provided. By default, the number of threads used is equal to the number of CPU cores available to the JVM.

5.1.1 Standard Input Builder Configurations

CloneWorks includes a number of bundled input builder configurations:

Type-1

Configuration File:	type1
Description:	For detecting Type-1 clones. To be used with the clone detector with a minimum similarity threshold of 100%.
Code-Fragment Processors:	-
Term Splitting:	By line.
Term Processors:	Joiner.
Details:	Each code fragment is represented by a single term equal its Type-1 normalized textual representation. Type-1 normalization includes the removal of comments, the normalization of the formatting (strict pretty-printing), and the normalization of the white-space.

Type-2 - Parametric Only

Configuration File:	type2_parametric
Description:	For detecting Type-2 "parametric" clones. These are clones that differ by only consistently renamed identifiers, in addition to Type-1 differences. To be used with the clone detector with a minimum similarity threshold of 100%.
Code-Fragment Processors:	-
Term Splitting:	By line.
Term Processors:	Joiner.
Details:	Each code fragment is represented by a single term equal its Type-2 (parametric) normalized textual representation. In addition to the Type-1 normalizations, the identifiers are consistently renamed.

Type-2 - Full

Configuration File:	type2
Description:	For detecting Type-2 clones. These are clones that differ by only their identifier names and literal values, in addition to any Type-1 differences. To be used with the clone detector with a minimum similarity threshold of 100%.
Code-Fragment Processors:	Blind Rename, Identifier Normalization.
Term Splitting:	By line.
Term Processors:	Joiner.
Details:	Each code fragment is represented by a single term equal its Type-2 normalized textual representation. In addition to the Type-1 normalizations, all identifiers are replaced by 'X' and all literals are replaced by '0'.

Type-3 - Conservative

Configuration File:	type3_conservative
Description:	For the detection of Type-3 clones with very-high precision and good recall. Clone detection should be executed with a minimum similarity threshold below 100%. Recommended: 70%.
Code-Fragment Processors:	-
Term Splitting:	By token.
Term Processors:	Filter Operators, Filter Separators.
Details:	Each code fragment undergoes just Type-1 before being split into language tokens. Then the operator and separator tokens are filtered. Code fragments are then represented as the bag, with duplicates, of keywords, primitives,and identifier tokens they contain. Clones are detected as the minimum ratio of these tokens that two code fragments share.

Type-3 - Aggressive

Configuration File:	type3_aggressive
Description:	For the detection of Type-3 clones with very-high recall and good precision. Clone detection should be executed with a minimum similarity threshold below 100%, we recommend 70%.
Code-Fragment Processors:	Blind Rename, Identifier Normalization.
Term Splitting:	By token.
Term Processors:	-
Details:	Each code fragment undergoes full Type-1 and Type-2 normalization before being split by line. Code fragments are then represented as the bag, with duplicates, of normalized code statements they contain. Clones are detected as the minimum ratio of these lines that two code fragments share.

5.2 Clone Detection (cwdetect)

The clone detector, cwdetect, usage is shown in Figure 2. The user must specify the file containing the code fragments to detect clones between, a file to output the clones to, and a minimum clone similarity threshold in range [0.0,1.0] for the Jaccard clone similarity metric. Optionally, minimum and maximum clone sizes can be specified.

If there are too many code fragments to fit in memory, then the partitioned partial indexes approach can be enabled with the "-p" flag, including the maximum number of code fragments in each partition. This number needs to be found experimentally. However, the first step is to load the code fragments into memory, so an out of memory exception should be encountered early if it is going to happen.

In our expirence, we found we can comfortably load 500000 code fragments into memory given a Java heap size of 8GB, with actual usage not exceeding approx. 6GB. This option is not needed for small inputs.

The "-ps" flag can be used to skip sorting the blocks. This is for use when the code-fragments are pre-sorted, or where the code fragments only have a single term (as is the case with our Type-1 and Type-2 configurations). Sorting the code fragment's terms is not the expensive part of clone detection, so its safer just to skip this flag unless you know your blocks are sorted properly and need the small performance boost by skipping sorting.

Clones are found in the output file in the following format:

```
10,40,50,20,60,70
```

This indicates a clone pair between lines 40-50 (inclusive) in source file 10 and lines 60-70 (inclusive) in source file 20. The mapping of IDs to source paths are in the fileids file produced by the input builder.

This is a compact format, but a more convenient format can be produced using the cwformat command.

Figure 2: cwdetect command-line usage

```
usage: cwdetect -i <file> -o <file> -s <ratio> [-p <num>] [-t <num>] [-mil <num>]
[-mal <num>] [-mit <num>] [-mat <num>] [-ps] [-idf]
Clone detection with CloneWorks.
-i,--input <file>                    File containing blocks produced by thrifty-builder.
-o,--output <file>                   File to output clones to.
-s,--min-similarity <ratio>          Minimum clone similarity.
-p,--partition-mode <num>            Run using index-partitioning mode with the
                                     specified maximum partition size in code blocks.
                                     Use when index size exhausts available memory.
-t,--num-threads <num>               Number of execution threads to use per task.
                                     Defaults to number of available cores.
-mil,--min-lines <num>               Minimum clone size in (original) source lines.
-mal,--max-lines <num>               Maximum clone size in (original) source lines.
-mit,--min-tokens <num>              Minimum clone size in (original, pre-processing)
                                     parsed tokens.
-mat,--max-tokens <num>              Maximum clone size in (original, pre-processing)
                                     parsed tokens.
-ps,--pre-sorted                     Indicates input blocks are pre-sorted, and
                                     GTF-sorting should be skipped.  This makes clone
                                     detection with pre-sorted blocks more efficient.
                                     If the blocks are not pre-sorted, or are sorted
                                     incorrectly, this can cause false negatives.
                                     Skipping sorting on pre-sorted blocks can improve
                                     performance.
```

Figure 3: cwformat command-line usage

```
usage: cwformat -f <path> -c <path> -o <path> -t <formater> [-v <options>]
CloneWorks-Format - Clone output formatter.
-f,--fileids <path>                 File containing the FileID<->Path mapping.
-c,--clones <path>                  File containing the detected clones output.
-o,--output <path>                  File to write the formatted clones to.
-t,--formater <formater>            The formatter to use.
-v,--formatter options <options>    A string containing the options for this formatter.
```

Formatter	Description	Required Options
csv	A CSV format.	-
xml	An XML format.	-
xml_withsrc	The XML format with the fragment's original source.	Path of the base source directory.

Table 1: Included Formatters

5.3 Clone Output Formatting (cwformat)

The output formatter, cwformat, is used to convert the clone output to a more convenient format. It supports multiple output formats, which we describe below, or you can add your own (see Section ??). Its usage is described in Figure 3.

The user must specify the file containing the mapping between the file IDs and source paths, the file containing the detected clones in their original output format, and a file to output the re-formatted clones to. The user must also specify the format they wish to use, and any formatting options it may require. CloneWorks includes the formatters listed in Table 1.

6 Advanced Usage

6.1 Input Builder Architecture

Please see the paper for details. The information will be added here in a concise form in a future version.

6.2 Input Builder Configuration

Configuration files define the code-fragment processors, term-splitting, and term processors to be used by the input builder. Configuration files are stored in the `config/` directory and refered to by name when executing `cwbuild`. The guide for the configuration file is shown in Figure 4. The code-fragment processors and term processors are executed in their given order, respectively.

6.3 Available Code-Fragment Processors

The included code-fragment processors are TXL-based, and parse the code fragments into abstract-syntax-trees (AST) to perform source transformations. Their prerequisite is that the code fragments must be in a state that is compatible with the TXL language grammar. It is recommended you use these processors before your custom processors if your processors may break the compatibility. Processors can have preconditions for use, but the framework is unable to check for these.

rename-consistent: This transformation normalizes identifier names by renaming them in a consistent way. The first unique identifier to appear in the code fragment is renamed to 'X1' everywhere it appears in the code fragment, the second unique identifier is renamed to 'X2', and so on. Can be used by adding the

Figure 4: Configuration File Guide

```
#
# Guide for creating configuration files.

############################
# Code-Fragment Processors #
############################

# Provide a list of code-fragment processors to apply to the code fragments.
# Applied in the order specified.  Specified as format:
#
#cfproc=name[ parameters]
#
# name - Name of executable in 'cfprocessors/'
# parameters - Parameters passed to the processor executable (in addition
#              to the standard ones).
#
#e.g.: 'transform=rename-blind'
#e.g.: 'transform=abstract literals'


##############
# Term Split #
##############

# How to split the code into terms.  Can be lines or language-tokens.  When
# splitting by line, use code-fragment processors to layout the code for the
# desired term definition.  By default, extraction performs strict
# pretty-printing and removes comments.  So split by line without any custom
# layout splits by pretty-printed code statements.

####################
# Token Processors #
####################

# Transformations/processing to apply on the terms.  Applied in the order
# specified.  Specified as format:
#
#tokproc=name[ initialization string]
#
# name - The name of the token processor class.
# initialization string - A string passed to teh class to initialize it.
#
#e.g.: 'termproc=FilterSeperators'
#e.g.: 'termproc=FilterOperators'
#e.g.: 'termproc=NGram 3'
```

Parameter	Name	Explanation
$1	Mode	Execution mode, ‘v’ = validate, ’r’ = run.
$2	InstallDir	Installation directory of CloneWorks (for finding dependencies).
$3	language	The source language of these code fragments.
$4	granularity	The code fragment granularity of these code fragments.
$5	param1	The first parameter from the configuration file for this processor.
$6	param2	The second parameter from the configuration file for this processor.
...	...	...

Table 2: Transformation Parameters

following command to the configuration file: transform=rename-consistent.

rename-blind: This transformation normalizes every identifier name to ‘X’. Can be used by adding the following command to the configuration file: transform=rename-blind.

abstract: This transformation replaces non-terminals with the name of their type. For example, the abstraction of literal will replace every literal value with ‘literal’. Can be used by adding the following command to the configuration file: transform=filter nonterminal_1 nonterminal_2 ... . See the TXL grammar for the language in txl/ for the names of the non-terminals.

filter: Replaced non-terminals with an empty string: ”” (they are filtered). Can be used by adding the following command to the configuration file: `transform=filter nonterminal\_1 nonterminal\_2 ....` See the TXL grammar for the language of interest in txl/ for the names of the non-terminals.

normalize-ifconditions This replaces the conditions in if statements with the string ‘expression’ For example, `if(x == 2)` is replaced by `if(expression)`. Can be used by adding the following command to the configuration file `transform=normalize-ifconditions`.

6.4 Adding a Custom Code-Fragment Processor

Transformations are implemented as executable or bash scripts placed in the `cfprocessors/` directory of CloneWorks. Then can be implemented to transform, normalize or filter the code fragments.

The code-fragment processors receives the code fragments over stdin in an xml-like format, like that shown in Figure 5. The code fragments will all be from the same original source file. It then outputs the transformed, normalized and/or filtered code fragments in the same format to stdout. Filtering can be done by omitting code fragments from the output. The processor can also split the code fragments into multiple code fragments, if desired. The processor can be done on the current state of the code fragment, and can access the original source file if additional analytics are required to perform the processor.

The processor also takes a number of command-line parameters, which are listed in Table 2. The first 4 parameters are always provided. Parameters after that are specific to the processor, and provided by the user in the configuration file.

The processors must support two execution modes, ‘validate’ and ‘run’.

The validate, ‘v’ mode is used to validate the processor. This is a sort of dry-run. The processor will be executed with the ‘v’ for the mdoe parameter, and all other parameters as expected during the experiment, but no input over stdin. The processor should examine these parameters and exit with a status ‘0’ if the language, granularity, and custom parameters (5 and on) are valid, or otherwise exit with status ‘1’ with an error message (a single line) output to stdout. This prevents executing a processor for a language, granularity or custom settings that are not supported, and give the user feedback on the issue (error message).

When the run, ‘r’, mode is specified, the processor is applied. If the processor fails, it should exit with a non-0 exit code. This will fail for all code fragments in the source file. Optionally, invalid ones can be discarded. Files that a processor fails for will be reported to the user.

Please see the provided processor in `cfprocessors/` for examples.

Figure 5: Code-Fragment Processor Input

```
<source file="src/CH/ifa/draw/util/FloatingTextField.java" startline="33" endline="35">
public FloatingTextField () {
fEditWidget = new JTextField (20);
}
</source>
<source file="src/CH/ifa/draw/util/FloatingTextField.java" startline="40" endline="42">
public void createOverlay (Container container) {
createOverlay (container, null);
}
</source>
<source file="src/CH/ifa/draw/util/FloatingTextField.java" startline="48" endline="54">
public void createOverlay (Container container, Font font) {
container.add (fEditWidget, 0);
if (font != null) {
fEditWidget.setFont (font);
}
fContainer = container;
}
</source>
<source file="src/CH/ifa/draw/util/FloatingTextField.java" startline="59" endline="61">
public void addActionListener (ActionListener listener) {
fEditWidget.addActionListener (listener);
}
</source>
<source file="src/CH/ifa/draw/util/FloatingTextField.java" startline="66" endline="68">
public void removeActionListener (ActionListener listener) {
fEditWidget.removeActionListener (listener);
}
</source>
```

6.5 Available Term Processors

The included term processors are shown in the termproc/src directory for review. These are already included in the distribution and can be used with the input builder.

Stemmer: This term processor performs stemming on the terms, as described by Martin Porter (`Inprogress...(seereadmefilesfornow).`). It has no parameters.
Usage: termproc=Stemmer

NGram: This performs a n-gram transformation over the terms. It takes the value of n as a parameter.
Usage: termproc=NGram n
Example: termproc=NGram 3

FilterOperators: This filters the terms that match operator tokens of the code fragment's given language.
Usage: termproc=FilterOperators

FilterOperators: This filters the terms that match seperator tokens of the code fragment's given language.
Usage: termproc=FilterSeperators

RemoveEmpty: This removes empty terms. Use if a previous transformation or term processor may produce empty terms.
Usage: termproc=RemoveEmpty

ToLowerCase: Normalzies the terms to lower-case characters only. *Usage:* termproc=ToLowerCase

TrimLeadingTrailingWhitespace: Removes any leading or trailing white-space from the terms. Use if a previous transformation or term processor may result in extraneous white-space which may prevent matching of equivalent same terms.
Usage: termproc=TrimLeadingTrailingWhitespace

SplitStrings: Splits string literals into its contained terms. Identifies string literals as terms that start adn end with a double quote, and splits on white-space. E.g., `"This is a    string"` results in terms `{This,is,a,string}`.
Usage: termproc=SplitStrings

NormalziedStrings: Normalizes string literals. Identifies string literals as terms that start and end wit ha double quote character, and replaces them with the term `""`.
Usage termproc=NormalizedStrings

Joiner: Combines all of the terms into a single term with a single space as a delimiter. E.g.: terms `{term1,term2,term3,term4}` becomes `{term1 term2 term3 term4}`.
Usage: termproc=Joiner

CombineAndHash: Like joiner, but replaces the joined terms by their hash value. Requires the hashing algorithm to be specified in a parameter, as specified below. Where algorithm must be one of the hashing algorithms supported by your JVM version (see `MessageDigest` javadoc).
Usage: termproc=CombineAndHash algorithm

6.6 Adding a Custom Term Processor

Write a Java class implementing the ITermProcessor interface provided. The class needs to be in the package "input.termprocessors". Your class must have a consturctor that takes a single string as a parameter and initializes the term processor using that string. Your class receives the parameter string provided in the input builder configuration file, or an empty string if no parameters are provided.

The interface requires the process() function, which takes a list of terms and must output a list of terms after the documented processing. Your term processor also receives the language, granularity, and term splitting used. These constants are available in the ITermProcessor. You can use these if your processing is depenant on these, or simply ignore them.

Compile your class and add its .class file to the bin/input/termprocessors/ directory. Your processor can now be found by reflection by the input builder. If your processor has additional dependencies, you must also add these to the bin/ directory in the appropriate class hierarchy directories. Or they can be added as a jar in the libs/ directory, requiring that the cwbuilder script is also modified to include the new libraries on the classpath.

See the examples in `termprocessors/` and `termprocessors/src/input/termprocessors`.

6.7 Adding a Custom Clone Output Formatter

Coming soon...