CloneWorks - Example Scenarios

Jeffrey Svajlenko
jeff.svajlenko@gmail.com

Chanchal K. Roy
chanchal.roy@usask.ca

November 2, 2016

Abstract

In this document we provide example scenarios for usage of CloneWorks, and how its input builder could be used to accomplish them. We begin with configurations for Type-1, Type-2 and Type-3 clone detection. Then we discuss a number of example scenarios and how CloneWorks can be used.

Contents

1 Type-1 Clone Detection

The user wants to detect the Type-1 clones, these are pairs of code fragments that are syntactically identical, when we ignore differences in white-space, layout and comments. The extraction of the code blocks will automatically apply these normalizations (removal of comments, and a strict pretty-printing to remove white-space and layout differences). There is then multiple ways to perform Type-1 detection, which are described below.

Option#1

Transformations:	-
Term Splitting:	Line
Term Processors:	Joiner
Similarity Threshold:	100%
Details:	Each code fragment is split by source line after strict pretty-printing and comment removal. These source-lines are then joined back together to make a single string. Each code fragment is represented by a set of a single term, which is its Type-1 normalized text. Code fragments are then 100% similar if they are Type-1 code fragment, or 0% otherwise. The partial index will immediately return *only* the Type-1 clones, but the terms are more expensive to store in memory.

Example Code Fragment:

```
public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fFigure = (TextHolder)dr.readStorable();
}
```

Example Representation: 1:public void read (StorableInput dr) throws IOException super.read (dr); fFigure = (TextHolder) dr.readStorable ();

Option#2

Transformations:	-
Term Splitting:	Line
Term Processors:	CombineAndHash
Similarity Threshold:	100%
Details:	Each code fragment is split by source line after strict pretty-printing and comment removal. The source-lines are then joined and the entire Type-1 normalized text of the code fragment is hashed. Each code fragment is then represented by the hash of its Type-1 normalized text. The code fragments are indexed by their hash, and the index returns just the code fragments with the same hash as a given code fragment. This can greatly reduce CPU and memory requirements compared to Option#1, but may report some false positives if hash collisions occur.

Code Fragment:

```
public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fFigure = (TextHolder)dr.readStorable();
}
```

Representation: `1:ababe263f7bb6e0e85ec3b2ff023480e`

Option#3

Transformations:	-
Term Splitting:	Line
Term Processors:	-
Similarity Threshold:	100%
Details:	Each code fragment is split by source line after strict pretty-printing and comment removal. The code fragments become represented by the set of Type-1 normalized code lines they contain. Type-1 clones are revealed by the code fragments whose representations have 100% similarity. The partial index will index only the most rare code-line in each code fragment, greatly reducing the number of code fragments compared to see if they are Type-1. However, some code fragments that are not Type-1 clones will be checked. But this is possibly lighter on memory since each unique source line only has to be stored in memory once.

Code Fragment:

```
public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fFigure = (TextHolder)dr.readStorable();
}
```

Representation:

```
1:public void read (StorableInput dr) throws IOException {
1:super.read (dr);
1:fFigure = (TextHolder) dr.readStorable ();
1:}
```

2 Type-2 Clone Detection

The user wants to detect the Type-2 clones. These are pairs of code fragments that are syntactically identical when we ignore differences in identifier names and literal values, in addition to the Type-1 differences. There is some disagreement on if Type-2 clones differ by just identifier names, or also literal values, and the extent of identifier changes (systematic identifier renaming or arbitrary identifier changes). This can be accomplished in the same way as Type-1 Clone Detection, with additional normalizations. Different combinations of normalizations can target different kinds of Type-2 clones.

To demonstrate these we use Option#3 strategy from the Type-1 clones. The Option#1 or Option#2 strategy could also be used by changing the term processor used to none (Option#1) or Combine And Hash (Option#2).

Parametric Clones

Transformations:	Rename-Consistent
Term Splitting:	Line
Term Processors:	Joiner
Similarity Threshold:	100%
Details:	Extraction of the code fragments normalized the Type-1 features. Additionally, consistent renaming of the identifiers is performed, normalizing any possible renaming of the identifiers. Then the parametric Type-2 clones become textually identical.

Code Fragment:

```
private void fireDrawingViewAddedEvent(final DrawingView dv) {
 final Object[] listeners = listenerList.getListenerList();
 DesktopListener dpl;
 DesktopEvent dpe = null;
 for (int i = listeners.length-2; i >= 0; i -= 2) {
 if (listeners[i] == DesktopListener.class) {
 if (dpe == null) {
 dpe = new DesktopEvent(MDIDesktopPane.this, dv);
 }
 dpl = (DesktopListener)listeners[i+1];
 dpl.drawingViewAdded(dpe);
 }
 }
}
```

Representation:

```
1:private void x1 (final x2 x3) {
1:final x4 [] x5 = x6.x7 ();
1:x8 x9;
1:x10 x11 = null;
1:for (int x12 = x5.x13 - 2;
1:x12 >= 0; x12 -= 2) {
1:if (x5 [x12] == x8.class) {
1:if (x11 == null) {
1:x11 = new x10 (x14.x15, x3);
1:x9 = (x8) x5 [x12 + 1];
1:x9.x16 (x11);
4:}
```

Parametric Clones + Identifiers

Transformations:	Rename-Consistent, Abstract Literals
Term Splitting:	Line
Term Processors:	Joiner
Similarity Threshold:	100%
Details:	Extraction of the code fragments normalized the Type-1 features. Additionally, consistent renaming of the identifiers is performed, normalizing any possible renaming of the identifiers, and the literal values are normalized, replacing each by 'literal'. Then the parametric + identifier Type-2 clones become textually identical.

Code Fragment:

```
private void fireDrawingViewAddedEvent(final DrawingView dv) {
    final Object[] listeners = listenerList.getListenerList();
    DesktopListener dpl;
    DesktopEvent dpe = null;
    for (int i = listeners.length-2; i >= 0; i -= 2) {
        if (listeners[i] == DesktopListener.class) {
            if (dpe == null) {
                dpe = new DesktopEvent(MDIDesktopPane.this, dv);
            }
            dpl = (DesktopListener)listeners[i+1];
            dpl.drawingViewAdded(dpe);
        }
    }
}
```

Representation:

```
1:private void x1 (final x2 x3) {
1:final x4 [] x5 = x6.x7 ();
1:x8 x9;
1:x10 x11 = null;
1:for (int x12 = x5.x13 - literal;
1:x12 >= literal; x12 -= literal) {
1:if (x5 [x12] == x8.class) {
1:if (x11 == null) {
1:x11 = new x10 (x14.x15, x3);
1:x9 = (x8) x5 [x12 + literal];
1:x9.x16 (x11);
4:}
```

Full Type-2 Clones

Transformations:	Rename-Consistent, Abstract Literals
Term Splitting:	Line
Term Processors:	Joiner
Similarity Threshold:	100%
Details:	Extraction of the code fragments normalized the Type-1 features. Additionally, arbitrary renaming of the identifiers is performed, normalizing all identifiers values, and the literal values are normalized, replacing each by 'literal'. Then the parametric + identifier Type-2 clones become textually identical.

Code Fragment:

```
private void fireDrawingViewAddedEvent(final DrawingView dv) {
    final Object[] listeners = listenerList.getListenerList();
    DesktopListener dpl;
    DesktopEvent dpe = null;
    for (int i = listeners.length-2; i >= 0; i -= 2) {
        if (listeners[i] == DesktopListener.class) {
            if (dpe == null) {
                dpe = new DesktopEvent(MDIDesktopPane.this, dv);
            }
            dpl = (DesktopListener)listeners[i+1];
            dpl.drawingViewAdded(dpe);
        }
    }
}
```

Representation:

```
1:private void x (final x x) {
1:final x [] x = x.x ();
1:x x;
1:x x = null;
1:for (int x = x.x - literal;
1:x >= literal; x -= literal) {
1:if (x [x] == x.class) {
1:if (x == null) {
1:x = new x (x.x, x);
1:x = (x) x [x + literal];
1:x.x (x);
4:}
```

3 Type-3 Clone Detection

The user wishes to detect Type-3 clones, which contain differences at the statement level. We suggest a conservative and aggressive detection strategies.

Conservative Type-3 Detection

Transformations:	-
Term Splitting:	Line
Term Processors:	Filter Separators, Filter Operators
Similarity Threshold:	[1-100]%, Recommended: 70%
Details:	The code fragments are split by language token, and the separator and operator tokens are filtered out. Code fragments are represented by the set of keyword, identifier and literal tokens they contain. Type-3 clones are detected by the minimum similarity of two code fragments' set of tokens. Has very good precision, with good recall.

Code Fragment:

```
private void fireDrawingViewAddedEvent(final DrawingView dv) {
    final Object[] listeners = listenerList.getListenerList();
    DesktopListener dpl;
    DesktopEvent dpe = null;
    for (int i = listeners.length-2; i >= 0; i -= 2) {
        if (listeners[i] == DesktopListener.class) {
            if (dpe == null) {
                dpe = new DesktopEvent(MDIDesktopPane.this, dv);
            }
            dpl = (DesktopListener)listeners[i+1];
            dpl.drawingViewAdded(dpe);
        }
    }
}
```

Representation:

```
1:fireDrawingViewAddedEvent
1:private
1:DrawingView
1:for
1:drawingViewAdded
2:dv
1:getListenerList
4:dpe
2:if
1:class
3:dpl
1:new
1:void
4:listeners
1:length
1:this
5:i
1:int
1:0
1:1
2:2
2:null
1:listenerList
3:DesktopListener
2:final
1:Object
2:DesktopEvent
1:MDIDesktopPane
```

Aggressive Type-3 Detection

Transformations:	Rename-Consistent, Abstract Literals
Term Splitting:	Line
Term Processors:	-
Similarity Threshold:	[1-100]%, Recommended: 70%
Details:	Type-1 and Type-2 normalizations are applied, and the code fragment is split by normalized line. This represents the code fragment as the set of code-statement patterns it contains. Type-3 clone detection is achieved by minimum similarity threshold. Provides very good recall with good precision.

Code Fragment:

```
private void fireDrawingViewAddedEvent(final DrawingView dv) {
    final Object[] listeners = listenerList.getListenerList();
    DesktopListener dpl;
    DesktopEvent dpe = null;
    for (int i = listeners.length-2; i >= 0; i -= 2) {
        if (listeners[i] == DesktopListener.class) {
            if (dpe == null) {
                dpe = new DesktopEvent(MDIDesktopPane.this, dv);
            }
            dpl = (DesktopListener)listeners[i+1];
            dpl.drawingViewAdded(dpe);
        }
    }
}
```

Representation:

```
1:private void x (final x x) {
1:final x [] x = x.x ();
1:x x;
1:x x = null;
1:for (int x = x.x - literal;
1:x >= literal; x -= literal) {
1:if (x [x] == x.class) {
1:if (x == null) {
1:x = new x (x.x, x);
1:x = (x) x [x + literal];
1:x.x (x);
4:}
```

4 Clones of API Usages

The user wishes to detect clones of API usage patterns. They can use the input builder to transform the code fragments into the set of API usage elements they contain. For example, they may use source transformations to extract the elements in a newline delimited list. They may extract the class names and function/member calls. Then split by newline to capture the elements in the set. Then they may use term processors to clean up the API elements. For example, replacing the parameters of a function call with just the number of parameters as a normalization. Executing clone detection would detect the code fragments that have similar API elements in a similar proportions as clones. Using different threshold, including different API elements, and different normalizations on the terms could change the results.

5 Cloning in Test Code

The user may want to detect Type-1, Type-2 or Type-3 clones of particular domains, for example just clones of test code. They can use the source transformations or term processors are filters for the code, performing analysis at either the code fragment or term level, and discarding any code fragment that does not fit the domain. For example, for test code, the code fragment could be analyzed for which APIs it uses, or language functions like asserts, to determine if it is a test code. Otherwise, one of the Type 1/2/3 strategies can be employed, adding these filters to the start of end of the transformations. This could be applied to any domain: clones of database access code, UI code, etc.

6 Type-4 Detection Additions

The source transformations can be used to apply significant structural normalizations to the code. For example, normalizing if/while/do-while loop structures to a common form before Type-1/2/3 detection (as described above). This could enable Type-4 extensions to the detection. Or similarly, TXL has been used to transform code between languages, which could improve cross-language clone detection.

7 Semantic Clone Detection with Topic Modeling

Lets say the user has developed a new system for topic modeling over code fragments. Their system was trained on a large collection of code, and can take a code fragment as input and output a list of topics that code fragment covers. They wish to use this topic list to perform clone detection of semantic clones, but do not want to develop a new clone detector for their topic list representation of code fragments.

They can take advantage of CloneWork's scalable and fast code fragment parser and clone detector. The user can implement a source transformation that sends each code fragment to their topic modeler server, and collect the topic list, converting the code fragment into a list of topics for clone detection. The user takes advantage of the input builder's code fragment parsing, and any other transformations required before topic modeling. The clone detector can then be used to efficiently and quickly detect all the clones that share some ratio of their measured topics.

In this scenario, the user used a source transformation to plug their topic modeling server into the CloneWorks workflow. The user can also use our input builder to help their topic modeler. The input builder could be used to parse, extract and transform their code fragments before training their topic modeler.