Anti-unification algorithms and their applications in program analysis

Peter E. Bulychev, Egor V. Kostylev, Vladimir A. Zakharov

Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, RU-119899, Russia

Perspectives of System Informatics, 2009
Outline

1. Anti-unification algorithms
2. Anti-unification for invariant generation
3. Anti-unification for clone detection
Anti-unification algorithms
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \rightarrow \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution

$$\theta = \{x/f(h(f(y, z), c), g(d, h(f(y, z), z)))\}$$

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta \eta$ – the composition of substitutions $\theta$ and $\eta$
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \rightarrow \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution

$$\theta = \{x/f(h(f(y, z), c), g(d, h(f(y, z), z)))\}$$

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta\eta$ – the composition of substitutions $\theta$ and $\eta$
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \to \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution

\[ \theta = \{ x/f(h(f(y, z), c), g(d, h(f(y, z), z))) \} \]

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta \eta$ – the composition of substitutions $\theta$ and $\eta$
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \rightarrow \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution

$$\theta = \{x/f(h(f(y, z), c), g(d, h(f(y, z), z)))\}$$

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta \eta$ – the composition of substitutions $\theta$ and $\eta$
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \rightarrow \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution
  
  $$\theta = \{x / f(h(f(y, z), c), g(d, h(f(y, z), z)))\}$$

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta \eta$ – the composition of substitutions $\theta$ and $\eta$
Basic notions

- $\mathcal{X}, \mathcal{Y}$ – sets of variables
- $\mathcal{F}$ – a set of functional symbols
- $\text{Term}[\mathcal{Y}, \mathcal{F}]$ – the set of terms over $\mathcal{Y}$ and $\mathcal{F}$
- $\theta : \mathcal{X} \rightarrow \text{Term}[\mathcal{Y}, \mathcal{F}]$ – a substitution

$$\theta = \{ x / f(h(f(y, z), c), g(d, h(f(y, z), z))) \}$$

- $E\theta$ – the application of a substitution $\theta$ to an expression $E$
- $\theta \eta$ – the composition of substitutions $\theta$ and $\eta$
Unification and anti-unification of substitutions

- quasi-order: \( \theta_1 \sqsubseteq \theta_2 \) iff there exists \( \xi \) such that \( \theta_2 = \theta_1 \xi \)

- \( \eta \) is the **most general instance** of substitutions \( \theta_1 \) and \( \theta_2 \) iff
  - \( \theta_1 \sqsubseteq \eta \) and \( \theta_2 \sqsubseteq \eta \)
  - if \( \theta_1 \sqsubseteq \eta' \) and \( \theta_2 \sqsubseteq \eta' \) then \( \eta \sqsubseteq \eta' \)
  \( \eta = \theta_1 \uparrow \theta_2 \) – unification

- \( \eta \) is the **most specific template** of substitutions \( \theta_1 \) and \( \theta_2 \) iff
  - \( \eta \sqsubseteq \theta_1 \) and \( \eta \sqsubseteq \theta_2 \)
  - if \( \eta' \sqsubseteq \theta_1 \) and \( \eta' \sqsubseteq \theta_2 \) then \( \eta' \sqsubseteq \eta \)
  \( \eta = \theta_1 \downarrow \theta_2 \) – anti-unification
Unification and anti-unification of substitutions

- quasi-order: $\theta_1 \sqsubseteq \theta_2$ iff there exists $\xi$ such that $\theta_2 = \theta_1 \xi$

- $\eta$ is the most general instance of substitutions $\theta_1$ and $\theta_2$ iff
  - $\theta_1 \sqsubseteq \eta$ and $\theta_2 \sqsubseteq \eta$
  - if $\theta_1 \sqsubseteq \eta'$ and $\theta_2 \sqsubseteq \eta'$ then $\eta \sqsubseteq \eta'$

  $\eta = \theta_1 \uparrow \theta_2$ – unification

- $\eta$ is the most specific template of substitutions $\theta_1$ and $\theta_2$ iff
  - $\eta \sqsubseteq \theta_1$ and $\eta \sqsubseteq \theta_2$
  - if $\eta' \sqsubseteq \theta_1$ and $\eta' \sqsubseteq \theta_2$ then $\eta' \sqsubseteq \eta$

  $\eta = \theta_1 \downarrow \theta_2$ – anti-unification
Unification and anti-unification of substitutions

- **quasi-order**: $\theta_1 \sqsubseteq \theta_2$ iff there exists $\xi$ such that $\theta_2 = \theta_1 \xi$

- $\eta$ is the *most general instance* of substitutions $\theta_1$ and $\theta_2$ iff
  - $\theta_1 \sqsubseteq \eta$ and $\theta_2 \sqsubseteq \eta$
  - if $\theta_1 \sqsubseteq \eta'$ and $\theta_2 \sqsubseteq \eta'$ then $\eta \sqsubseteq \eta'$

  $\eta = \theta_1 \uparrow \theta_2$ – unification

- $\eta$ is the *most specific template* of substitutions $\theta_1$ and $\theta_2$ iff
  - $\eta \sqsubseteq \theta_1$ and $\eta \sqsubseteq \theta_2$
  - if $\eta' \sqsubseteq \theta_1$ and $\eta' \sqsubseteq \theta_2$ then $\eta' \sqsubseteq \eta$

  $\eta = \theta_1 \downarrow \theta_2$ – anti-unification
Representation of substitutions

common representation: sets of labeled directed trees

compact representation: reduced dags

the size $N(\theta)$ of substitution $\theta$ – the number of nodes in the reduced dag.
The complexity of anti-unification: upper bound

Theorem

Let $\theta'$ and $\theta''$ be a pair of substitutions represented by reduced dags. Then reduced dag for $\eta = \theta' \downarrow \theta''$ can be computed in time $O(n \log n)$, where $n = N(\eta)$.

Corollary

The most specific template of substitutions $\theta'$ and $\theta''$ represented by reduced dags can be computed in time $O(n^2 \log n)$, where $n = \max(N(\theta'), N(\theta''))$. 
The complexity of anti-unification: upper bound

**Theorem**

Let $\theta'$ and $\theta''$ be a pair of substitutions represented by reduced dags. Then reduced dag for $\eta = \theta' \downarrow \theta''$ can be computed in time $O(n \log n)$, where $n = N(\eta)$.

**Corollary**

The most specific template of substitutions $\theta'$ and $\theta''$ represented by reduced dags can be computed in time $O(n^2 \log n)$, where $n = \max(N(\theta'), N(\theta''))$. 
The complexity of anti-unification: lower bound

**Theorem**

Suppose that $\mathcal{F}$ contains a functional symbol of arity $m > 1$. Then there exists an infinite sequence of pairs of substitutions $(\theta'_i, \theta''_i), \ i \geq 1$, such that

$$\frac{1}{6} N(\theta'_i) \times N(\theta''_i) \leq N(\theta'_i \downarrow \theta''_i).$$

**Corollary**

If $\mathcal{F}$ contains a functional symbol of arity $m > 1$ then time complexity of the anti-unification problem for two substitutions represented by reduced dags is $\Omega(n^2)$, where $n = \max(N(\theta'), N(\theta''))$. 
The complexity of anti-unification: lower bound

**Theorem**

Suppose that $\mathcal{F}$ contains a functional symbol of arity $m > 1$. Then there exists an infinite sequence of pairs of substitutions $(\theta'_i, \theta''_i), \ i \geq 1$, such that

$$\frac{1}{6} N(\theta'_i) \times N(\theta''_i) \leq N(\theta'_i \downarrow_\mathcal{F} \theta''_i).$$

**Corollary**

If $\mathcal{F}$ contains a functional symbol of arity $m > 1$ then time complexity of the anti-unification problem for two substitutions represented by reduced dags is $\Omega(n^2)$, where $n = \max(N(\theta'), N(\theta''))$. 
Anti-unification for invariant generation
Abstract model of sequential program

**assignment statement** $e$ is an expression $\ell_{in} : v \leftarrow t : \ell_{out}$ where
- $v$ – variable
- $t$ – term
- $\ell_{in}, \ell_{out}$ – non-negative integers, those are called **entry and exit points**

A non-deterministic abstract program $\Pi$ is a pair $\langle L, E \rangle$ where
- $L$ – a finite set of non-negative integers (**program points**)
- $E$ – a finite set of assignment statements

0 is the **entry point** of $\Pi$
Abstract model of sequential program

Assignment statement $e$ is an expression $\ell_{in} : v \leftarrow t : \ell_{out}$ where

- $v$ – variable
- $t$ – term
- $\ell_{in}, \ell_{out}$ – non-negative integers, those are called entry and exit points

Non-deterministic abstract program $\Pi$ is a pair $\langle L, E \rangle$ where

- $L$ – a finite set of non-negative integers (program points)
- $E$ – a finite set of assignment statements

0 is the entry point of $\Pi$
Example of program

program test(z)
    x1 = z;
    z = z+1;
    x2 = z;
    while x2==x1+1 do
        x1 = z;
        if prime(z)
            then z = x2+1
        else x1=2*z; z=2*x2+1
        fi;
        x2 = z
    od;
end.

program Π₀:
0 : x₁ ⇐ z : 1,
1 : z ⇐ f(z, c₁) : 2,
2 : x₂ ⇐ z : 3,
2 : x₂ ⇐ z : 7,
3 : x₁ ⇐ z : 4,
4 : z ⇐ f(x₂, c₁) : 6,
4 : x₁ ⇐ g(c₂, z) : 5,
5 : z ⇐ f(g(c₂, x₂), c₁) : 6,
6 : x₂ ⇐ z : 7,
6 : x₂ ⇐ z : 3.
A trace $tr$ of a program $\Pi$ is a finite sequence of statements

$$tr = e_1, e_2, \ldots, e_m$$

such that

- $in(e_1) = 0$
- $out(e_i) = in(e_{i+1})$ for every $i$, $1 \leq i < m$.

The trace $tr$ leads to the point $out(e_m)$.

$Tr_{\Pi}(\ell)$ – the set of all traces of a program $\Pi$ leading to a point $\ell$
Semantics of programs

- \( M = \{ D_M, \bar{f}_1, \ldots, \bar{f}_k, = \} \) — a first-order structure with equality
- \( \sigma : \mathcal{V} \rightarrow D_M \) — data state of program with set of variables \( \mathcal{V} \)
- \( t[\sigma] \) — the value of term \( t \) in the data state \( \sigma \)

Let

- \( tr = e_1, e_2, \ldots, e_m \) — a trace such that
  
  \[ e_i = \ell_{i-1} : v_i \leftarrow t_i : \ell_i, \ 0 < i \leq m \]

- \( \sigma_0 \) — a data state
  
  then

\[ \sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_n \]

is the run for \( tr \) and \( \sigma_0 \) if

\[ \sigma_i = \sigma_{i-1}[v_i \leftarrow t_i[\sigma_{i-1}]] \] for every \( i, 0 < i \leq m \).
Semantics of programs

- $M = \{ D_M, \bar{f}_1, \ldots, \bar{f}_k, = \}$ – a first-order structure with equality
- $\sigma : \mathcal{V} \rightarrow D_M$ – data state of program with set of variables $\mathcal{V}$
- $t[\sigma]$ – the value of term $t$ in the data state $\sigma$
- let
  - $tr = e_1, e_2, \ldots, e_m$ – a trace such that $e_i = \ell_{i-1} : v_i \leftarrow t_i : \ell_i$, $0 < i \leq m$
  - $\sigma_0$ – a data state

then

$$\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_n$$

is the run for $tr$ and $\sigma_0$ if

$$\sigma_i = \sigma_{i-1}[v_i \leftarrow t_i[\sigma_{i-1}]]$$ for every $i, 0 < i \leq m$
Invariants

- A first-order formula $\Phi(v_1, \ldots, v_n)$ is an \textit{M-invariant} of a program $\Pi$ at a point $\ell$ if it holds in model $M$ for every run in $\Pi$.
- $\Phi(v_1, \ldots, v_n)$ is a \textit{strong invariant} if it is an $M$-invariant for every model $M$.
- An invariant $\Phi$ is \textit{the most specific strong invariant} if the formula $\Phi \rightarrow \Psi$ is valid for every strong invariant $\Psi$.
- $\Phi(v_1, \ldots, v_n)$ is an \textit{equality invariant} if it has the form
  $$\exists y_1 \ldots \exists y_k (v_1 = t_1 \land v_2 = t_2 \land \cdots \land v_n = t_n).$$
Invariants

- A first-order formula $\Phi(v_1, \ldots, v_n)$ is an \textit{M-invariant} of a program $\Pi$ at a point $\ell$ if it holds in model $M$ for every run in $\Pi$.
- $\Phi(v_1, \ldots, v_n)$ is a \textbf{strong invariant} if it is an $M$-invariant for every model $M$.
- An invariant $\Phi$ is the \textit{most specific strong invariant} if the formula $\Phi \rightarrow \Psi$ is valid for every strong invariant $\Psi$.
- $\Phi(v_1, \ldots, v_n)$ is an \textit{equality invariant} if it has the form
  \[ \exists y_1 \ldots \exists y_k (v_1 = t_1 \land v_2 = t_2 \land \cdots \land v_n = t_n). \]
Invariants

- A first-order formula $\Phi(v_1, \ldots, v_n)$ is an $M$-invariant of a program $\Pi$ at a point $\ell$ if it holds in model $M$ for every run in $\Pi$.
- $\Phi(v_1, \ldots, v_n)$ is a strong invariant if it is an $M$-invariant for every model $M$.
- An invariant $\Phi$ is the most specific strong invariant if the formula $\Phi \rightarrow \Psi$ is valid for every strong invariant $\Psi$.
- $\Phi(v_1, \ldots, v_n)$ is an equality invariant if it has the form

$$\exists y_1 \ldots \exists y_k (v_1 = t_1 \land v_2 = t_2 \land \cdots \land v_n = t_n).$$
Invariants

- A first-order formula \( \Phi(v_1, \ldots, v_n) \) is an \( M \)-invariant of a program \( \Pi \) at a point \( \ell \) if it holds in model \( M \) for every run in \( \Pi \).
- \( \Phi(v_1, \ldots, v_n) \) is a strong invariant if it is an \( M \)-invariant for every model \( M \).
- An invariant \( \Phi \) is the most specific strong invariant if the formula \( \Phi \rightarrow \Psi \) is valid for every strong invariant \( \Psi \).
- \( \Phi(v_1, \ldots, v_n) \) is an equality invariant if it has the form
  \[
  \exists y_1 \ldots \exists y_k (v_1 = t_1 \land v_2 = t_2 \land \cdots \land v_n = t_n).
  \]
Substitutions associated with program

we associate:

- with every statement $e = \ell_{in} : v_i \leftarrow t : \ell_{out}$ a substitution
  \[
  \theta_e = \{ v_1/v_1, \ldots, v_{i-1}/v_{i-1}, v_i/t, v_{i+1}/v_{i+1}, \ldots v_n/v_n \}
  \]

- with every trace $tr = e_1, e_2, \ldots, e_m$ a substitution
  \[
  \eta_{tr} = \theta_{e_m} \ldots \theta_{e_2} \theta_{e_1}
  \]

- with every point $\ell$ of a program $\Pi$ a substitution
  \[
  \theta_{\Pi, \ell} = \downarrow_{tr \in Tr_\Pi(\ell)} \eta_{tr}
  \]
Substitutions and invariants

**Theorem**

Let $\Pi = \langle L, E \rangle$ be a program and $\ell$ be a point of $\Pi$. Suppose that

$$\theta_{\Pi,\ell} = \{ v_1/t_1, v_2/t_2, \ldots, v_n/t_n \}.$$

Then the formula

$$\Phi_{\Pi,\ell} = \exists y_1 \ldots \exists y_k (v_1 = t_1 \land v_2 = t_2 \land \cdots \land v_n = t_n),$$

where $\{y_1, \ldots, y_k\}$ is the set of all variables occurred in the terms $t_1, t_2, \ldots, t_n$, is the most specific strong equality invariant at the point $\ell$. 
Theorem

For every program \( \Pi = \langle L, E \rangle \), the set of substitutions \( \{ \theta_{\Pi, \ell} : \ell \in L \} \) is the least solution to the system

\[
\Omega(\Pi) : \begin{cases} 
\Theta_{\ell} = \downarrow_{e \in E, out(e) = \ell} \theta_{e} \Theta_{in(e)}, \; \ell \in L, \ell \neq 0, \\
\Theta_{0} = \{ v_{1}/y_{1}, \ldots, v_{n}/y_{n} \}.
\end{cases}
\]

This theorem relies upon left distributivity of composition of substitutions over anti-unification

\[
\eta(\theta_{1} \downarrow \theta_{2}) = \eta \theta_{1} \downarrow \eta \theta_{2}.
\]
Example of program

```plaintext
program test(z)
    x1 = z;
    z = z+1;
    x2 = z;
    while x2==x1+1 do
        x1 = z;
        if prime(z) then z = x2+1
        else x1=2*z; z=2*x2+1
        fi;
        x2 = z
    od;
end.
```

```plaintext
program Π₀:
0: x₁ ← z : 1,
1: z ← f(z, c₁) : 2,
2: x₂ ← z : 3,
2: x₂ ← z : 7,
3: x₁ ← z : 4,
4: z ← f(x₂, c₁) : 6,
4: x₁ ← g(c₂, z) : 5,
5: z ← f(g(c₂, x₂), c₁) : 6,
6: x₂ ← z : 7,
6: x₂ ← z : 3.
```

Peter E. Bulychev, Egor V. Kostylev, Vladimir A. Zakharov

Anti-unification algorithms and their applications
Example of invariant

\[ \Omega(\Pi_0) : \]

\[ \begin{align*}
\Theta_0 &= \{x_1/y_1, x_2/y_2, z/y_3\} \\
\Theta_1 &= \{x_1/z, x_2/x_2, z/z\} \Theta_0 \\
\Theta_2 &= \{x_1/x_1, x_2/x_2, z/f(z, c_1)\} \Theta_1 \\
\Theta_3 &= \{x_1/x_1, x_2/z, z/z\} \Theta_2 \downarrow \{x_1/x_1, x_2/z, z/z\} \Theta_6 \\
\Theta_4 &= \{x_1/z, x_2/x_2, z/z\} \Theta_3 \\
\Theta_5 &= \{x_1/g(c_2, z), x_2/x_2, z/z\} \Theta_4 \\
\Theta_6 &= \{x_1/x_1, x_2/x_2, z/f(x_2, c_1)\} \Theta_4 \downarrow \\
&\quad \downarrow \{x_1/x_1, x_2/x_2, z/f(g(c_2, x_2), c_1)\} \Theta_5 \\
\Theta_7 &= \{x_1/x_1, x_2/z, z/z\} \Theta_2 \downarrow \{x_1/x_1, x_2/z, z/z\} \Theta_6
\end{align*} \]

obtained invariant for the point 3:

\[ \Phi = \exists y (x_1 = y \land x_2 = y + 1 \land z = y + 1) \]

\(\Phi\) implies \(x_2 = x_1 + 1\)

conclusion: the source program never terminates.
Anti-unification
for clone detection
Definitions of clone

Intuitive definition.
Fragments of code which are similar to each other are called code clones.

Formal definition.
Suppose that $\rho$ is a distance and $|\cdot|$ is a measure on the set of code fragments. A pair $E_1, E_2$ of code fragments is $(d_1, d_2)$-clone if

- $\rho(E_1, E_2) \leq d_1$,
- $|E_1| \geq d_2$ and $|E_2| \geq d_2$. 
Motivation for clone detection

- 5% - 20% of code in software systems are clones
- Why do programmers produce clones?
  - development strategy
  - maintenance benefits
  - overcoming underlying limitations
  - cloning by accident
- Why is the presence of code clones bad?
  - Errors in the original must be fixed in every clone.
Anti-unification distance

We consider:

- sequences of statements as code fragments (represented by AST)
- the number of different leaves in AST of $E = E_1 \downarrow E_2$ (anti-unification distance) as $\rho(E_1, E_2)$
- the number of different leaves in AST of $E$ as $|E|$

example of AST:
Clone example

```cpp
x = a;
y = f(x, i);
cout << y;
```

```cpp
x = a + b;
y = f(x, j);
cout << y;
```

```
=  
/ 
x  a

=  
/ 
y  f

cout
```

```
=  
/ 
x  +

=  
/ 
y  f

cout
```

```
=  
/ 
x  i

=  
/ 
y

```

```
=  
/ 
x  j

=  
/ 
y  f

+  
/ 
/ 
/ 
/ 
/ 
/ 
/ 
a  b

```

Peter E. Bulychev, Egor V. Kostylev, Vladimir A. Zakharov

Anti-unification algorithms and their applications
The sketch of the clone detection algorithm
The sketch of the clone detection algorithm

1. Partition similar statements into clusters
The sketch of the clone detection algorithm

1. Partition similar statements into clusters
2. Find pairs of code fragments whose corresponding elements (statements) belong to the same clusters
The sketch of the clone detection algorithm

1. Partition similar statements into clusters
2. Find pairs of code fragments whose corresponding elements (statements) belong to the same clusters
3. Refine the identified pairs of code fragments by checking their structural similarity
Clusterization

First pass: definition of clusters’ centers.

1. Every statement $E'$ is compared with the templates $E_C = \downarrow_{E \in C} E$ of all existing clusters.

2. If the distance $\rho(E', E_C)$ is below some predefined threshold then $E'$ is put into $C$ and the cluster’s template is updated to $E_C \downarrow E'$.

3. If no such cluster was found then $E'$ forms a new cluster.

Second pass: partition to clusters.

1. Every statement $E'$ is put into the cluster $C$ such that $\rho(E', E_C)$ is minimal (we don’t update $E_C$).
Clone Digger

- is written in Python
- is provided under the GPL license and hosted on Source Forge (http://clonedigger.sf.net)
- is able to detect clones in: Python, Java 1.5, Lua, Javascript
- writes the information on found clones to HTML with a highlighting of differences
Comparision with CloneDR$^{\text{TM}}$

- CloneDR$^{\text{TM}}$ is a commercial tool, working on the AST level.
- We tested evaluation version, which reports only top 9 clone items.
- Clone Digger is able to detect all the clones which were reported by CloneDR$^{\text{TM}}$.
- However some of the clones detected by Clone Digger could not be detected by CloneDR$^{\text{TM}}$ in principle.
Comparision with CloneDR™

<table>
<thead>
<tr>
<th>Project</th>
<th>Size (loc)</th>
<th>Clones(%)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CD</td>
<td>CloneDR</td>
</tr>
<tr>
<td>netbeans-javadoc</td>
<td>14k</td>
<td>21.48</td>
<td>14.82</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>146k</td>
<td>14.24</td>
<td>18.09</td>
</tr>
</tbody>
</table>
Thank You