Abstract

This paper presents the design and implementation of CD-Form (Clone Detector based on FORmal Methods), a tool targeted at the detection of Type-2 clones in Java code. CD-Form is based on a novel approach for detecting code clones. The methodology adopted performs the analysis on Java bytecode and not on the original Java source. The bytecode is transformed into CCS (Calculus of Communicating Systems) processes, which are successively checked for equivalence. After a thorough description of the methodology used for detecting clones, the design of the tool is presented. The results obtained by evaluating sample Java codes are validated by comparing them to those obtained by a state-of-the-art tool for clone detection.

Keywords: Clone detection; Formal Methods; CCS

1. Introduction

Reusing code fragments by copying and pasting with minor modifications is customary in software development. As a result, software systems often contain sections of code that are similar. These sections are commonly referred to as code clones. Clone detection has been recognized as an important issue in software analysis. In fact, many software engineering tasks may require the extraction of syntactically or semantically similar code fragments. These activities include program maintenance (if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug), program reuse (it is useful to know of similar codes to be reused), program understanding (clones may carry domain knowledge) code quality analysis (fewer clones may mean better code quality), plagiarism detection (the presence of clones may mean stolen code), code readability (by identifying duplicated code and refactoring it, the size of code is reduced).

In practice, manual clone detection is infeasible for large software systems, and so automatic support is necessary. Over the last decade, multiple code clone detection techniques and tools have been proposed [1]. In this paper, we present a novel technique, based on a formal method approach, which has proven to be
effective for detecting source code clones by analyzing the Java bytecode that is output of Java compilers. Detecting clones in the compiled code and not in the source has pros and cons [2]. On the one hand, different but semantically equivalent source codes can lead to the same bytecode, showing up trivially the presence of clones. On the other hand, sometimes a small change in the source can produce completely different binaries, hiding the reuse of the same code fragments. Sometimes the analysis of the compiled code is the only alternative, because the source is not available; after all, Java bytecode has become the common way to distribute programs and to execute them through the web. In light of all the above, we think that the analysis of the bytecode is an option, even when the source code is available.

Formal methods are powerful techniques for specifying and verifying complex systems. Among the formal methods that have been developed over the past three decades, there is a small class of methods, collectively called process algebra, that find their roots in algebra. They represent a mathematically rigorous framework for modeling systems, describing their evolution by operational semantics. Moreover, they often provide observational mechanisms that make it possible to identify through behavioral equivalences those systems that are externally indistinguishable.

There are many examples of process algebras. The Calculus of Communicating Systems (CCS) of Milner [3] is one of the best known process algebras. From a textual CCS specification it is possible to generate the corresponding labeled transition system, which can be successively used for equivalence checking. Equivalence checking is the process of determining whether two systems are equivalent to each other according to some mathematically-defined notion of equivalence. Equivalence checking is typically used to verify if a system design conforms to its high-level “service specification”. For example, if the observable behaviour of a communication protocol is identical to that of a perfect communication channel that delivers all messages in order, then it would be justifiable to deem the protocol correct.

In this paper we will show how to use equivalence checking to detect whether two fragments of code are clones, transforming them into CCS processes and then checking if they are equivalent using a suitably-defined equivalence relation. One of the advantages of this procedure based on formal methods is that once the CCS model is generated, formal verification tools such as [4] can be used also for verifying safety and liveness program properties. Properties are expressed using temporal logic [5].

The distinctive features of the clone detection approach proposed in this paper are the identification of clones on Java bytecode and the use of formal methods. As far as the use of compiled code is concerned, it should be pointed out that previous work by Baker and Manber [6] has pioneered the clone analysis of Java bytecode. Recently, an attempt of performing clone detection on assembler code has been made by Davis and Godfrey in [7]. In [2], Keivanloo et al. exploit the bytecode representation for detecting semantically similar methods in the Java bytecode. On the other hand, the exploitation of process algebra-based formal methods for checking code equivalence is a novel approach,
which, in our knowledge, has never been used before. Even if the use of a mathematical code description for detecting relatively simple code similarities may seem overkill, we think that it is a viable solution, due to the availability of formal verification tools. Furthermore, it opens a wide field of opportunities to researchers for code understanding and documentation purposes.

Following up the description of the clone detection methodology, preliminarily presented in [8, 9], this paper deals with the details of the design and implementation of CD-Form, an efficient, practical and scalable tool based on our formal approach. We have used this tool to analyze a sample of Java systems belonging the well-known set used in [10]. We have evaluated both the CD-Form performance, i.e., the speed at which the detector returns its results, and its scalability, i.e., the extent to which the detector can manage increasingly large systems. The results of these tests are discussed in Section 5.

The remainder of the paper is organized as follows. Section 2 is a review of the basic concepts of CCS and of clone detection basics, while Section 3 describes our methodology. In Section 4 CD-Form, the tool implementing our approach, is presented, and in Section 5 the experimental results we obtained are reported and discussed. Finally, comparisons with related work are made in Section 6, and our conclusions are presented in Section 7.

2. Preliminaries

In this section, after introducing the basic definitions of code clones, we present the Calculus of Communicating Systems (CCS) [3], the process algebra we have adopted for our clone detection prototype.

2.1. Clone detection

Software clone detection is an active field of research. In the following a basic introduction to clone detection terminology from [11] is reported.

Definition 2.1 (Code Fragment). A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or simply a sequence of statements.

Definition 2.2 (Code Clone). A code fragment CF2 is a clone of another code fragment CF1 if they are similar according to some given definition of similarity, that is, f(CF1) = f(CF2) where f is the similarity function. Two fragments that are similar to each other form a clone pair (CF1; CF2); when several fragments are similar, they form a clone class or clone group.

Definition 2.3 (Clone Types). The types of clones commonly recognized in the literature [1] are the following:

• Type-1. Identical code fragments, except for variations in whitespace, layout and comments.
• **Type-2.** Syntactically identical fragments, except for variations in identifiers, literals, types, whitespace, layout and comments.

• **Type-3.** Copied fragments with slight modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

• **Type-4.** Fragments that perform the same computation but are implemented by different syntactic variants.

### 2.2. Process algebra: CCS

Historically, process algebras have developed as formal descriptions of complex computer systems, and in particular of those involving communicating, concurrently executing components. The crucial idea in the definition of Process Algebras is the algebraic structure of the concurrent processes. This uses a state-based approach with labeled transitions, where states and transitions correspond to processes and actions, respectively. There are many examples of Process Algebras. Milner’s Calculus of Communicating Systems (CCS)\[3\] is one of the most well known process algebras, and is largely used for modeling concurrent and distributed systems. Below we present only a brief overview of the main features of CCS. Readers unfamiliar with CCS are referred to [12, 3] for further details.

The syntax of processes is the following:

\[
p ::= \text{nil} \quad \text{nil} \\
| \alpha.p \quad \text{prefix} \\
| p + p \quad \text{summation} \\
| p|p \quad \text{composition} \\
| p\mid L \quad \text{restriction} \\
| p[f] \quad \text{relabeling} \\
| x \quad \text{constant}
\]

where

- \(\alpha\) ranges over a finite set of actions \(A = \{\tau, a, \pi, b, \bar{\pi}, \ldots\}\). Input actions are labeled with “non-barred” names, i.e., \(a\), while output actions are “barred”, i.e., \(\bar{\pi}\). The action \(\tau \in A\) is called internal action. \(\tau\) actions allow some level of abstraction in the description of the processes, since they can hide an arbitrarily complex sequence of actions whose details are kept private. The set of visible actions \(V\) is defined as \(A - \{\tau\}\). Each action \(l \in V\) (resp. \(\bar{l} \in V\)) has a complementary action \(\bar{l}\) (resp. \(l\)). Process may interact whenever a process is prepared to perform some action \(l\) and the other is prepared to perform the complementary action \(\bar{l}\).

- The restriction set \(L\), in the processes of the form \(p\mid L\), is a set of actions such that \(L \subseteq V\). 


• The relabeling function \( f \), in processes of the form \( p[f] \), is a total function, \( f: \mathcal{A} \rightarrow \mathcal{A} \), such that the constraint \( f(\tau) = \tau \) is respected.

• The constant \( x \) ranges over a set of constant names: each constant \( x \) is defined by a constant definition \( x \overset{\text{def}}{=} p \), where \( p \) is called the body of \( x \).

We denote the set of processes by \( \mathcal{P} \). The standard operational semantics [3] is given by a relation \( \rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P} \). We give the semantics for CCS by induction over the structure of processes.

• Nil. \( \text{nil} \) represents a process that can do nothing. There is no rule for \( \text{nil} \) since it cannot evolve.

• Prefix. The process \( \alpha.p \) can perform the action \( \alpha \) and thereby become the process \( p \). This is expressed by the rule Act:

\[
\begin{align*}
\text{Act} & \quad \alpha.p \xrightarrow{\alpha} p
\end{align*}
\]

• Summation. The process \( p + q \) is a process that non-deterministically behaves either as \( p \) or as \( q \). This is expressed by the rules Sum\textsubscript{1} and Sum\textsubscript{2}:

\[
\begin{align*}
\text{Sum\textsubscript{1}} & \quad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} & \text{Sum\textsubscript{2}} & \quad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}
\end{align*}
\]

• Composition. The operator \( | \) expresses parallel composition. \( p \) and \( q \) may act independently: if the process \( p \) can perform \( \alpha \) and become \( p' \), then \( p|q \) can perform \( \alpha \) and become \( p'|q \), and similarly for \( q \). This is expressed by the rules Par\textsubscript{1} and Par\textsubscript{2}:

\[
\begin{align*}
\text{Par\textsubscript{1}} & \quad \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q} & \text{Par\textsubscript{2}} & \quad \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p'|q'}
\end{align*}
\]

Furthermore, \( p \) and \( q \) may also be engaged in a communication whenever they are able to simultaneously execute the complementary actions. In this case we say that a handshake has occurred. Handshake corresponds to an internal communication: the two processes exchange information between them and the action is \( \tau \). This is expressed by the rule Com:

\[
\begin{align*}
\text{Com} & \quad \frac{p \xrightarrow{\tau} p', q \xrightarrow{\tau} q'}{p|q \xrightarrow{\tau} p'|q'}
\end{align*}
\]

• Restriction. If \( L \) is a set of visible actions, \( p \\backslash L \) is a process that behaves as \( p \) except that it cannot perform any of the actions (as well as the corresponding complementary actions) lying in \( L \) externally, although each
pair of these complementary actions can be performed for communication internally. This is expressed by the rule Res:

$$\text{Res} \quad \frac{p \xrightarrow{\alpha} p'}{p \xrightarrow{\alpha \in \pi \not \in L}}$$

- **Relabeling.** The operator $[f]$ expresses the relabeling of actions. If $p$ can perform $\alpha$ and become $p'$, then $p[f]$ can perform $f(\alpha)$ and become $p'[f]$. This is expressed by the rule Rel:

$$\text{Rel} \quad \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$$

- **Constant definition.** The behavior of the process $x$ ($x \overset{\text{def}}{=} p$) is that of its definition $p$ as expressed by the rule Con:

$$\text{Con} \quad \frac{p \xrightarrow{\alpha} p'}{x \overset{\text{def}}{=} p}$$

A (labeled) transition system is a quadruple $T = (S, A, \rightarrow, s)$, where $S$ is a set of states, $A$ is a set of transition labels (actions), $s \in S$ is the initial state, and $\rightarrow \subseteq S \times A \times S$ is the transition relation. If $(s, \alpha, s') \in \rightarrow$, we write $s \xrightarrow{\alpha} s'$. If $\delta \in A^*$ and $\delta = \alpha_1 \ldots \alpha_n, n \geq 1$, we write $s \xrightarrow{\delta} s'$ to mean $s \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s'$. Moreover, $s \xrightarrow{\lambda} s$, where $\lambda$ is the empty sequence. Given $s \in S$, with $R(p) = \{ s' | s \xrightarrow{\delta} s' \}$ we denote the set of the states reachable from $s$ by $\rightarrow$. Given a CCS process $p$, the standard transition system for $p S(p) = (R(p), A, \rightarrow, p)$. Given a CCS process $p$, the standard transition system for $p$ is $S(p) = (R(p), A, \rightarrow, p)$.

Note that, with abuse of notation, we use $\rightarrow$ for denoting both the operational semantics and the transition relation among the states of the transition system.

Many equivalence relations have been defined on CCS processes; they are based on the notion of bisimulation between states of the related transition systems. In the following we consider the well-known weak equivalence, which describes how processes (i.e., systems) match each other’s behavior. In order to define the weak equivalence, we first introduce the following transition relation between processes.

**Definition 2.4.** Let $p$ and $q$ be two CCS processes. We write $p \xrightarrow{\tau} q$ if and only if there is a (possibly empty) sequence of $\tau$ actions that leads from $p$ to $q$. If the sequence is empty, then $p = q$. For each action $\alpha$, we write $p \xrightarrow{\alpha} q$ if there are processes $p'$ and $q'$ such that $p \xrightarrow{\alpha} p'$ and $q' \xrightarrow{\tau} q$. For each action $\alpha$, we use $\hat{\alpha}$ to stand for $\epsilon$ if $\alpha = \tau$, and for $\alpha$ otherwise.
Thus $p \xrightarrow{\alpha} q$ holds if $p$ can reach $q$ by performing an $\alpha$ action, possibly preceded and followed by sequences of $\tau$ actions. For example $a.\tau.nil \xrightarrow{a} \tau.nil$ and $\alpha.nil \xrightarrow{\alpha} \tau.nil$.

The idea underlying the following definition of weak equivalence is that an action of a process can be matched by a sequence of actions from the other that has the same “observational content” (i.e., ignoring $\tau$ actions) and leads to a state that is equivalent to that reached by the first process.

**Definition 2.5.** (weak bisimulation, weak equivalence). Let $p$ and $q$ be two CCS processes.

- A weak bisimulation, $B$, is a binary relation on $P \times P$ such that $p B q$ implies:

  (i) if $p \xrightarrow{\alpha} p'$ then $\exists q' : q \xrightarrow{\alpha} q'$ for some $q'$ such that $p' B q'$; and

  (ii) if $q \xrightarrow{\alpha} q'$ then $\exists p' : p \xrightarrow{\alpha} p'$ for some $p'$ such that $p' B q'$

- $p$ and $q$ are weak equivalent ($p \approx q$) iff there exists a weak bisimulation $B$ containing the pair $(p,q)$.

### 3. The Methodology

In this section we present the methodology underlying our CD-Form tool. The two distinctive features of this methodology are the use of formal methods (to the Authors’ knowledge, never used before) and the detection on Java bytecode and not on the source code. In practice, from the Java bytecode we derive CCS processes, which are successively checked for equivalence. The goal of our methodology is to capture Type-1 and Type-2 clones at method (function) granularity. Since there is no general agreement about these definitions, we formulate a proposal of a working definition of Java Type-2 clones. It is worth pointing out that Type-1 clones can be detected fairly easily, especially at the bytecode level, where the discrepancies in whitespaces and layout have been eliminated by the compiler. On the other hand, Type-2 clone detection requires a more substantial effort, since all information about types must be ignored, together with variations in identifiers and literals. Thus, in defining our methodology, the following problems have been tackled:

1. Type-2 clones in Java: an extended definition
2. Bytecode-level clone detection
3. Transformation of the code into an internal representation
4. Clone detection process
After dealing with the first problem by extending the definition of Type-2 to the Java language, we analyze issues related to performing clone detection on the bytecode and not directly on the Java code. The third problem, which has great impact on the accuracy of results, is solved by translating the bytecode representation into CCS processes. Finally, the clone detection process is discussed. This process applies equivalence checking techniques from CCS to detect clone pairs and returns only clone pairs as result. In order to reduce the amount of output data and improve their readability, we can aggregate clone pairs into classes of mutually equivalent fragments.

In the following, the four issues are examined in more detail.

3.1. Type-2 clones in Java: an extended definition

Definition 2.3 introduces a taxonomy that is just a starting point to understand clone detection techniques. In fact, it is too generic and ambiguous to be used as a specification for a concrete clone detection tool. To this aim, a more specific clone definition must be used, taking also into account the target programming language. Our clone definition starts from an interpretation of Definition 2.3. The rationale behind this taxonomy is to define different classes of clones, with growing levels of dissimilarity. The Type-1 class is made up of code fragments with small variations in code formatting (spaces and comments). The Type-2 class includes fragments with differences that do not alter the syntactic structure of the code. In Type-3 clones, deeper modifications are allowed. These involve changing, adding or removing statements. Finally, Type-4 clones can also have variations of the whole syntactic structure, provided that the computation performed is the same.

In this spirit, we have contextualized the original definition to the Java language, thus obtaining an “extended” Type-2 definition. The rationale is to allow variations of all the language elements that do not alter the syntactic structure of the code, and are not significant as far as the code logic is concerned. In our work, Type-2 Java clones are made of syntactically identical fragments, except for the variations described in Definition 2.3 for Type-1 and Type-2 clones (identifiers, literals, types, whitespaces, layout and comments), plus differences in:

- Modifiers: visibility modifiers (public, private, protected), abstract, final, native, static, synchronized, transient, volatile and strictfp.
- Casting operators, e.g., different casts or no cast at all, as ((Type1) expression vs. (Type2) expression vs. expression).
- Annotations, as @Deprecated, @Override, or custom annotations.
- Syntactic variants of compound identifiers, as for example id (simple reference) vs. this.id (member reference) vs. ClassName.this.id (class-qualified reference) vs. packageName.ClassName.id (fully qualified reference, ClassName.id (for static members).
3.2. Bytecode-level clone detection

Our methodology exploits the bytecode representation of Java programs produced by a Java compiler. Performing clone detection on the bytecode and not directly on the Java code has the disadvantage that it is necessary to process the possibly different Java bytecode generated by different compilers. In fact, the JVM specification does not impose a specific translation of source to bytecode instructions, but leaves a certain amount of freedom to compilers, provided that the generated code is compliant with the given Java version. As it has been documented in literature [13], for real applications there actually exist differences in the generated bytecode; the amount of different code varies on an application basis. This may impose some limitations to our approach, in the (unlikely) situation in which the objective is to compare Java binaries compiled with different compilers and the source code is not available. On the other hand, our design choice has several advantages:

- independence of the source programming language (i.e., minor changes to the Java syntax become insignificant);
- clone detection without decompilation even when source code is lacking;
- straight clone detection of merely similar constructs that may “look different” when expressed as source code, avoiding source normalization;
- ease of parsing a lower-level code.

As mentioned above, detecting clones at bytecode level introduces subtle issues that should be taken into account when the results are compared to those obtained at source level. First of all, source code formatting becomes irrelevant at the bytecode level, as the compiler already removes variations in whitespace and comments. This entails that Type-1 clones are automatically detected. Additional source normalizations are not necessary to remove superficial differences such as changes in statement bracketing (i.e., \texttt{if (a) b=2;} vs. \texttt{if (a) \{b=2;\}}). Aside from formatting, the most significant issue is due to the addition of code (or alteration of the original one) that could possibly be performed by the compiler, leading to mismatches between the source-level clones and the bytecode-level clones. Specifically:

- different fragments of source (i.e., not clones) can be translated to bytecode sequences that are detected as clones;
- fragments that are clones at source level are translated into different sequences of bytecode, and not recognized as clones.

As an example of the first issue, let us consider Figure 1. In class \texttt{One}, the programmer has explicitly invoked the superclass constructor, which is absent in class \texttt{Two}. It turns out that these two fragments are not Type-2 clones of each other (instead, the added statement makes them Type-3 clones). However, at bytecode level, the two fragments are equal. The second issue is shown in
Figure 2: the two call methods look exactly the same at source code level, and would be recognized as clones by any Type-2 clone detector. Anyway, the method from class Inner requires the loading on the stack of an instance of the enclosing type Top, and this causes the insertion of an extra `aload_0` instruction in the method translation. This does not happen for the method from class Other. As a consequence, there is a difference in bytecodes that can deceive the detector.

```
public One(java.lang.String);
Code:
0: aload_0
1: invokespecial #1;
4: aload_0
5: aload_1
6: putfield #2;
9: return
```

(b) Bytecode for One constructor

```
public Two(java.lang.String);
Code:
0: aload_0
1: invokespecial #1;
4: aload_0
5: aload_1
6: putfield #2;
9: return
```

d) Bytecode for Two constructor

3.3. Transformation of the code into an internal representation

We use as internal representation the CCS language. Thus, CCS specifications are generated from Java bytecode. This is obtained by defining a Java bytecode-to-CCS transform operator $T$. The function $T$ directly applies to the Java bytecode of a program and translates it into CCS process specifications. The objective of $T$ is to avoid the construction of “expensive” data structures such as Abstract Syntax Trees (ASTs) or Program Dependence Graphs (PDGs) while retaining their accuracy for clone detection. The function $T$ is defined for each instruction of the Java bytecode. In the following, a Java-byte program $P$ is a sequence $c$ of instructions, numbered starting from address 0; $\forall i \in \{0, \ldots, |c|\}$, and $c[i]$ is the instruction at address $i$, where $|c|$ denotes the length of $c$. All Java bytecode instructions have been translated in CCS; below we will show only a few, just to give the reader the flavor of the approach followed.

**Instruction:** $c[i] = \text{goto } j$

$$T(i) = x_i \overset{\text{def}}{=} \text{gotoj}, x_j$$

The instruction $c[i] = \text{goto } j$ is translated into a CCS process $x_i$ that performs the action $\text{gotoj}$ and then jumps to the instruction $j$, corresponding to the CCS
class Top{
    class Inner{
        public void aMethod(String arg){
            System.out.println(arg);
        }
        public void call(){
            Inner inner = new Inner();
            inner.aMethod("hello");
        }
    }
}

class Other{
    public void aMethod(String arg){
        System.out.println(arg);
    }
    public void call(){
        Other other = new Other();
        other.aMethod("hello");
    }
}

(a) Original Source Code
(b) Bytecode for call method

class Top{
    class Inner{
        public void aMethod(String arg){
            System.out.println(arg);
        }
        public void call(){
            Inner inner = new Inner();
            inner.aMethod("hello");
        }
    }
}

class Other{
    public void aMethod(String arg){
        System.out.println(arg);
    }
    public void call(){
        Other other = new Other();
        other.aMethod("hello");
    }
}

(c) Original Source Code
(d) Bytecode for call method

Figure 2: Same source code translated to different bytecode
process $x_j$.

**Instruction:** $c[i] = \text{tstore } x$

$$T(i) = x_i \text{ def } = \text{store } x_{i+1}$$

Since Type-2 clones are considered, each $\text{tstore } x$ instruction is translated, regardless of the type $t$ and of the name of the variable $x$, as $\text{store }$ followed by the constant process $x_{i+1}$ representing the CCS translation of the successive instruction.

**Instruction:** $c[i] = \text{iinc } x \ k$

$$T(i) = x_i \text{ def } = \text{load } . \text{const } . \text{add } . \text{store } x_{i+1}$$

$i\text{inc}$ increments the integer held in the local variable $x$ by $k$. It should be noted that there is no equivalent instruction to increment variables of other types. For example, the Java compiler uses a sequence of four opcodes ($\text{load } \text{const } \text{add } \text{store}$) to increment a float variable. As our objective is to spot Type-2 clones, a uniform translation of an instruction across different types must be provided. So the $i\text{inc}$ opcode is translated exactly the same way as the increment of a float variable.

**Instruction:** $c[i] = \text{if} \ _\text{cond } j$

The bytecode produced by compiling a Java $\text{if}$ statement is not unique, but depends on the types of the involved variables/values. One family of branching-on-condition opcodes is used to compare integer values, and another to compare float values.

In the case of integers, there is a set of branching on condition opcodes that perform integer comparisons against zero ($\text{ifeq }, \text{ifne}, \ldots$) and, depending on the result, branches or proceeds in sequence. Another set ($\text{if_icmpeq}, \text{if_icmpne}, \ldots$) is used to compare two integers, popping them off the top of the stack, comparing them against one another, and branching/not branching according to the result of the comparison. In one case or another (i.e., both for comparisons of an integer against zero and comparisons of two integers) the CCS translation is the same, as follows:

$$T(i) = x_i \text{ def } = \text{if } \_\text{cond } tt \ . \text{if } \_\text{cond } ff \ . x_{i+1} + \text{if } \_\text{cond } tt \ . x_j$$

The true (resp. false) condition is represented by the CCS action $\text{if } \_\text{cond } tt$ (resp. $\text{if } \_\text{cond } ff$), while

$$\text{cond } \in \{ \text{eq }, \text{icmeq }, \text{icmpne }, \ldots \}.$$  

In the case of floats, the comparison-and-branch is carried out in two steps. First, a compare ($\text{fcmp }, \text{fcmpg }, \ldots$) is executed. The int value that represents the result of the comparison ($0$ for equal to, $1$ for greater than, and $-1$ for less than) is pushed on the stack. Then a branch on condition opcode ($\text{ifeq }, \text{ifne},$
the same used for integer comparisons against zero) is executed to force the actual branch. For example, the Java statement:

\[
\text{if} \ (a < b) \ s_1 \ \text{else} \ s_2
\]

is translated into Java bytecode using only \text{if} \text{icmpge} when \(a\) and \(b\) are integer, while it is translated using the pair \text{fcmpg}, \text{ifge} when \(a\) and \(b\) are float.

As in the case of the increment, the choice to detect Type-2 clones imposes the use of the same CCS translation regardless of the type. In other words, we have to make equivalent the CCS translation of \text{if} \text{icmpge} to that of the pair \text{fcmpg}, \text{ifge}. Thus, when transforming in CCS a float branching on condition opcode, we relabel the action representing the opcode \text{fcmpg} to \(\tau\) and we make similar the CCS translation of \text{if} \text{icmpge} and \text{if} \text{ge} using the relabeling operator. Suppose that \(x_2\) is the CCS process corresponding to the instruction \(s_2\), and \(x_1\) is the CCS process corresponding to the instruction \(s_1\). The CCS transformation of the instruction in 1 (with \(a, b\) integer) is:

\[
x_i \overset{\text{def}}{=} \text{if} \text{icmpge}_{\text{ff}}.x_1 + \text{if} \text{icmpge}_{\text{tt}}.x_2
\]

The CCS transformation of the instruction in 1 (with \(a\) and \(b\) float) is:

\[
x_i \overset{\text{def}}{=} \tau.(\text{if} \text{ge}_{\text{ff}}.x_1 + \text{if} \text{ge}_{\text{tt}}.x_2)[f]
\]

where \(f = [\text{if} \text{icmpge}_{\text{ff}}/\text{if} \text{ge}_{\text{ff}}, \text{if} \text{icmpge}_{\text{tt}}/\text{if} \text{ge}_{\text{tt}}]\)

Thus, the general CCS translation of float branching on condition opcodes is:

\[
T(i) = x_i \overset{\text{def}}{=} \tau.(\text{if} \text{cond}_{\text{ff}}.x_i+1 + \text{if} \text{cond}_{\text{tt}}.x_j)[f]
\]

where \(f = [\text{if} \text{icmpcond}_{\text{ff}}/\text{if} \text{cond}_{\text{ff}}, \text{if} \text{icmpcond}_{\text{tt}}/\text{if} \text{cond}_{\text{tt}}]\)

The true (resp. false) condition is represented by the CCS action \text{if} \text{cond}_{\text{tt}} (resp. \text{if} \text{cond}_{\text{ff}}), while \text{cond} \in \{\text{ge}, \text{ne}, \ldots\}.

We illustrate each step of our methodology using the following simple running example.

\textbf{Example 3.1 (Running example).}

Let us consider the two Java methods of Table 1 (a) which are Type-2 clones of each other. The Java bytecode of both methods is shown in Table 1 (b). As we can see, the two methods do not look exactly the same at bytecode level. Applying our methodology, the CCS specifications of Table 2 are generated from Java bytecode. More precisely, the process \(X\) is the CCS transformation of \texttt{method1}, while \(Y\) is the CCS process corresponding to \texttt{method2}.
public class Example {
    public int method1(int a, int b){
        int n=0;
        if(a<b) n+=a;
        return n;
    }
    public double method2(double a, double b){
        double n=0;
        if(a<b) n+=a;
        return n;
    }
}

(a) Original Source Code

public class Example extends java.lang.Object{
    public Example();
    Code:
    0: aload_0
    1: invokespecial #8; //Method java/lang/Object."<init>":()V
    4: return
    public int method1(int, int);
    Code:
    0: iconst_0
    1: istore_3
    2: iload_1
    3: iload_2
    4: if_icmpge 11
    7: iload_3
    8: iload_1
    9: iadd
   10: istore_3
   11: iload_3
   12: ireturn
    public double method2(double, double);
    Code:
    0: dconst_0
    1: dstore 5
    3: dload_1
    4: dload_3
    5: dcmpeq
    6: ifge 15
    9: dload 5
   11: dload_1
   12: dadd
   13: dstore 5
   15: dload 5
   17: dreturn

(b) Bytecode

Table 1: Running example
3.4. Clone detection process

In our approach, the algorithm used to identify clones is equivalence-based. Once we have the CCS processes of the Java bytecode fragments, we can use known equivalence relations to determine clone detection. Depending on the type of clones to be discovered, different equivalence relations can be considered. Since our focus is on Type-2 clones, weak equivalence has been used. Intuitively, two CCS processes, representing Java bytecode fragments, are considered to be weak equivalent if they are indistinguishable from the viewpoint of an external observer interacting with them. Although CCS has been historically used for modeling concurrent systems, in this paper we exploit its algebraic structures mainly for equivalence checking. A factor that may lead to unacceptable high execution times is related to the number of equivalence checks. Currently our methodology returns only clones of methods. This implies that \( O(n^2) \) equivalence checks are necessary for \( n \) methods. This complexity,
which is practically too expensive, can be reduced avoiding any comparison between methods that are clearly different. In practice, only “reasonable” clone candidates are checked. A number of optimizations is introduced in the core algorithm to reduce the number of comparisons:

- **Clone granularity:** first of all, it is possible to impose a threshold on the minimum number of bytecode instructions in a method.
- **Different parameters:** we can avoid comparing two methods if they have a different number of parameters, as this is a statement change not allowed in Type-2 clones.
- **Number of actions:** again for the definition of Type-2 clones, we can avoid comparing two methods if their CCS translation contains a different number of visible actions, as they can’t be clones.
- **Number of distinct actions:** by maintaining a set of the actions in a method, we can again avoid comparison if the number of distinct visible actions is different.
- **Different sort:** we can compare the set of actions (the sort of a CCS process), and no two methods with different sort can be Type-2 clones. For further optimization, this comparison can be performed after a check of the number of distinct actions: as this number also indicates the cardinality of the sort, we can compare the sorts only when that number is the same, avoiding the costlier element-wise set comparison when not needed.
- **Transitivity:** Type-2 clones form an equivalence relation, unlike Type-3 clones, where statements can be added or removed. A further reduction of the quadratic complexity is obtained exploiting the transitive property of the weak equivalence. Thus, if we know that \( p \sim q \) and \( q \sim r \), we can avoid to check whether \( p \sim r \), since it is certainly true.

**Example 3.2 (Running example (continued)).**

Reconsider Example 3.1. After generating the CCS processes, we use weak equivalence to determine whether \texttt{method1} and \texttt{method2} are clones of each other. First of all, we check if the two CCS process \( X \) and \( Y \) corresponding to the two methods are reasonable candidates. It turns out that both \( X \) and \( Y \) have the same number of visible actions, which is equal to 6. Moreover, \( X \) and \( Y \) have the same sort (i.e., the set \{ \texttt{push}, \texttt{store}, \texttt{if icmpge}, \texttt{if icmpgett}, \texttt{add}, \texttt{allreturn} \}). Note that, by definition of the relabelling function \( f \), the action \texttt{ifge} (resp. \texttt{ifgett}) of the CCS process \( Y \) is relabelled with \texttt{if icmpge} (resp. \texttt{if icmpgett}). Thus, \( X \) and \( Y \) are checked for equivalence, since they are considered good candidates. It turns out that \( X \) is weak equivalent to \( Y \), since the following set of pairs:

\[
B = \{(X, Y), (X_0, Y_0), (X_1, Y_1), (X_2, Y_3), (X_3, Y_4), (X_4, Y_5),
(X_7, Y_9), (X_8, Y_11), (X_9, Y_12)(X_{10}, Y_{13})(X_{11}, Y_{15}), (X_{12}, Y_{17})\}
\]

is a weak bisimulation relating the states of \( S(X) \) with those of \( S(Y) \). This implies that \texttt{method1} is Type-2 clone of \texttt{method2}.
4. The CD-Form clone detector

Figure 3: Architecture of CD-Form

The methodology presented in the previous section has been exploited for the implementation of the CD-Form tool, whose architecture is shown in Figure 3. CD-Form is a hybrid Java/C++ system which represents an evolution of the initial prototype described in [9]. The workflow of the two tools is the same, in that the bytecode that resides in a class folder or in JAR files is fed to a custom parser, based on the Apache Commons Bytecode Engineering Library (BCEL)\(^1\). The parsed Java methods are successively translated into CCS processes following the methodology described in the previous section. Then CD-Form invokes the CCS equivalence checker. The results of the equivalence check are fed back into the tool, which, in case of a match, produces a clone pair. Multiple clone pairs whose fragments are mutually clones are clustered into clone classes. Finally, CD-Form outputs all the retrieved clone pairs and classes.

In [9], the prototype has been shown to yield high accuracy both of precision (i.e., returns few false positives) and recall (i.e., returns few false negatives). However, the execution time for small projects was of the order of ten minutes, making the use of the tool impractical in most real-world cases. In light of the above, CD-Form has been largely redesigned to face two fundamental challenges: performance, the speed at which the detector can return its results, and scalability, the extent to which the detector can manage increasingly large systems. The following subsection will describe the most relevant modifications performed with the aim of turning an initial rapid research prototype into an efficient, practical and scalable tool.

4.1. Implementation

In the prototype presented in [9], the long execution times were mostly due to three factors:

- The low scalability of the clone detection algorithm: this is due to the \(O(n^2)\) complexity of the algorithm, where \(n\) is the number of methods in the system under test.
- The low scalability of the equivalence checking process: formal methods, and most of the current formal verification tools, have scalability issues as

\(^1\)http://commons.apache.org/bcel/
the size of the system representation grows. In terms of our methodology, this means that we face scalability issues as the size of the CCS processes grows, that is as the number of lines in the methods of the system under test grows larger.

- The overhead to invoke an external executable (the Concurrency Workbench of New Century (CWB-NC) [4], a formal verification environment offering tool for checking equivalences) from Java code.

To face the first problem, in Section 3.4 we have proposed a number of techniques to reduce the number of comparisons, the positive effect of which will be discussed in the experiments of Section 5.

Regarding the second problem, recently several solutions to scale up formal methods and thus also equivalence checking have been proposed, including compositional techniques [14, 15], abstraction [16, 17] and Binary Decision Diagrams (BDD) [18]. In [19] an efficient approach based on heuristic search techniques to check equivalences has been defined. In general, a heuristic search strategy [20] uses an evaluation function to determine the order in which the nodes in the search graph are selected for expansion. The evaluation function measures the distance to a goal node based on a heuristic function which gives the estimated shortest distance from the given node to a goal node. In [19] the equivalence checking has been formalized as a search problem on AND/OR graphs. Moreover, an efficient equivalence checking procedure has been proposed, defining a heuristic function that suggests to expand first the states that offer the most promising way to deduce that two systems are not equivalent. This makes it possible to avoid the exhaustive exploration of the global state graph of the two systems when they are not equivalent. One of the authors of this paper has contributed to the design of Grease\textsuperscript{2} (GREedy Algorithm for System Equivalence), a C++ tool supporting the heuristic approach to check equivalence of CCS processes. The use of Grease on a sample of CCS processes has shown a significant reduction of both state-space size and time, compared to traditional equivalence checking algorithms.

The third problem is more strictly related to the implementation. In the original prototype, invoking an external executable program from Java code required the creation of a system process for every equivalence check. Moreover, the CCS representation of the processes was passed to the CWB-NC in the form of files, incurring additional overhead for writing the files and reading it in the CWB-NC. To obtain higher performance and scalability, the Grease checker has been integrated in the CD-Form architecture, and so equivalence checking can be performed with no invocation of external programs, as our old tool did. The use of the Java Native Interface (JNI) makes it possible for CD-Form to communicate with Grease without introducing any overhead due to process creation (by loading the library in the same process of the Java Virtual Machine).

\textsuperscript{2}http://www2.ing.unipi.it/~a080224/grease/
4.2. Validation

As a first validation of the approach proposed, we adopted the idea behind the framework proposed in [1]. Starting from an original code fragment, we generated all kinds of code variations that can lead to a Type-1 or Type-2 clone, verifying that the detector is able to recognize the changed fragments as clones of the original. For brevity’s sake, the whole process will not be presented here. The reader will find all the details in [1]. CD-Form achieved 100% precision and recall in recognizing all the Type-1/Type-2 variations. Of course, this check is not sufficient for a complete validation. The next section will describe experiments that have been conducted on real-world software systems.

5. Experimental results

To test the current implementation of CD-Form, we set up a comparison with an existing clone detection tool, NiCad 3.5 [21, 22]. According to its authors, NiCad is a hybrid parser-based/text comparison clone detection system. NiCad adopts a two-step process (plus an optional one) to find clones. In the first phase, a language-sensitive parser extracts functions or blocks of code. These potential clones are successively compared by means of a diff algorithm based on the search for the longest common subsequence (LCS). An optional intermediate step applies code normalization, processing the potential clones for renaming (blindly or consistently substituting identifiers), filtering or abstraction. The last two techniques allow to discard or to abstract in the comparison, respectively, a subset of the nonterminals of the language-specific grammar employed by the parser. NiCad has been chosen as reference because it is a mature and flexible tool [23], which allows for customization of both the language grammar and of the detection process, making it possible to detect Type-2 Java clones. NiCad can be configured to recognize the clones as defined in Section 3.1 if one allows for:

- extraction of potential clones at the function level;
- variations in whitespaces, layout and comments: this is a standard feature of NiCad, as in the first step its parser pretty-prints the lines and strips the comments;
- variations in identifiers: this requires the configuration of NiCad for blind or consistent renaming. According to the more general interpretation of Definition 2.3, we opted for blind renaming.
- variations in types: this can be obtained by configuring NiCad so as to abstract the type modifier grammar nonterminal;
- variations in literals: obtained by configuring NiCad to abstract the literal grammar nonterminal;
- no adding or removal of statement: this can be obtained by configuring NiCad with a 0.0 threshold;
• variations in compound identifiers: to achieve this, first a custom nonterminal `compound_identifier` is created to match all the expressions as `id` or `this.id` or `this.field.id`. Then all the expressions in the grammar that refer to basic identifiers are modified to refer to this new compound identifier. Finally, this nonterminal is added to the list of abstracted nonterminals;

• variations in modifiers: achieved by filtering the `modifier` grammar nonterminal;

• variations in casting operators: to achieve this, first a custom nonterminal `cast_operator` is created in the grammar to capture cast operations during parsing. Then this nonterminal is added to the list of filtered nonterminals;

• variations in annotations: achieved by filtering the `annotation` grammar nonterminal.

To compare the results of NiCad with those obtained by our tool, we chose to compare clone pairs. Another option would be to compare directly clone classes, but this may lead to unfairly low precision and recall, if classes are only partially overlapped. Consider a situation in which a detector finds a class with 4 fragments (for a total of 6 pairs) and the other only finds 3 fragments (for a total of 3 pairs) in the same class. If we compare the classes, no match is generated. Using directly the clone pairs produces a 50% correspondence, allowing a finer-grained comparison.

A problem with the comparison between the two tools is the different meaning of granularity. Both NiCad and CD-Form make it possible to specify the desired granularity. However, while NiCad refers to the number of (pretty-printed) source lines, we refer to the number of bytecode instructions. In [9], we used a granularity of 1 for both tools, in order to achieve a correct validation and to retain a good precision. The drawback is that a high number of scarcely meaningful clones are detected (e.g., getters and setters methods). In the present work, we set up the tools to take into account only clones longer than a reasonable threshold.

To achieve this result, we leveraged the NiCad feature that makes it possible to produce, in addition to the detected clones, the list of all the functions in the system. Using this list, it is possible to build a map that associates every Java method with the number of pretty-printed lines it contains. Then, we run the two tools with a minimum threshold of 5 pretty-printed lines for NiCad and a minimum granularity of 3 bytecode instructions for CD-Form. This guarantees that in terms of length, our tool will consider a superset of the comparisons that NiCad performs (as no function of 5 lines can produce less than 3 instructions of bytecode). Finally, before comparing the CD-Form results with the ones by NiCad, we remove from the CD-Form clones those where the fragments (methods) are smaller than 5 source pretty-printed lines (by exploiting the map described above). Using this approach, we try to preserve validation, precision
and meaningfulness of the clones. The only drawback is that our tool will perform comparisons that will be discarded (those for methods with more than 3 bytecode instructions but less than 5 source code lines). This may penalize it, as far as the time needed for detection is concerned.

In [9], we presented preliminary results obtained on two small-scale Java projects (less than 10 KLOC of code), with running times of about 10 minutes. Thanks to optimizations devised for CD-Form implementation, the tool performance has greatly improved, making it now possible to test all the Java systems in the well-known set used in [10]. In the mentioned paper, four systems of different size are chosen:

- NetBeans Javadoc (referred to as netbeans-javadoc in the following), containing 19 KLOC;
- Eclipse Ant (eclipse-ant in the following), containing 35 KLOC;
- Eclipse JDTCore (eclipse-jdtcore in the following), containing 148 KLOC;
- Java 2 SDK 1.4.0 Swing components (j2sdk1.4.0-javax-swing in the following), containing 204 KLOC.

We analyzed the four Java systems mentioned above, using NiCad with a suitably-tuned configuration as a reference. We considered true positives the clones found by both detectors, false positives the clones found only by our tool, and false negatives the clones found only by NiCad. Since precision and recall are derived from these basic metrics, they also adopt NiCad as reference.

The results obtained on the four software systems are reported in Table 3. For the netbeans-javadoc project, CD-Form achieved a 100 percent recall and a precision of almost 97 percent. The additional clones found by CD-Form mainly consist of fragments with differences in parenthesization, not considered clones by NiCad. In the case of eclipse-ant, CD-Form misses 7 clones found by NiCad, due to occurrences of the final keyword causing the production of different bytecode. For the project eclipse-jdtcore, precision turn out to be lower, because of a limitation in the current implementation of CD-Form that causes the translation of expressions like x++ and assignments like x=x+1 to be the same, leading to a higher number of clones detected. The recall is instead over 95 percent, with differences mainly due to extra arguments to constructors inserted by the compiler (as described in Section 3.2), not yet recognized by CD-Form. The same issue manifests itself in the last project, j2sdk1.4.0-jdk-swing, leading to a 95 percent recall. The loss of precision here is mainly due to different parenthesization, as described above.

In light of the above, overall the results by NiCad and CD-Form are very close. Table 4 shows the execution times of CD-Form and NiCad for the four projects. The test machine is the same for both detectors, based on an Intel Xeon Processor E3-1225 with 4gb of RAM running a Ubuntu Linux 12.04. CD-Form analyzes every project in less than 5 minutes, a reasonable amount of time for systems of over two hundred thousand lines of code.
Table 3: Experimental clone detection results

<table>
<thead>
<tr>
<th>Project</th>
<th>$T_p$</th>
<th>$F_n$</th>
<th>$F_p$</th>
<th>$P$</th>
<th>$R$</th>
<th>$Fm$</th>
</tr>
</thead>
<tbody>
<tr>
<td>netbeans-javadoc</td>
<td>443</td>
<td>0</td>
<td>14</td>
<td>96.94%</td>
<td>100.00%</td>
<td>98.44%</td>
</tr>
<tr>
<td>eclipse-ant</td>
<td>396</td>
<td>7</td>
<td>28</td>
<td>93.40%</td>
<td>98.26%</td>
<td>95.77%</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>6404</td>
<td>296</td>
<td>1485</td>
<td>81.17%</td>
<td>95.58%</td>
<td>87.79%</td>
</tr>
<tr>
<td>j2sdk1.4.0-jdk-swing</td>
<td>23206</td>
<td>1019</td>
<td>453</td>
<td>98.10%</td>
<td>95.79%</td>
<td>96.93%</td>
</tr>
</tbody>
</table>

$T_p$: Clones found by both detectors  
$F_n$: Clones found only by NiCad  
$F_p$: Clones found only by CD-Form  
$P$: Precision, defined as $T_p/(T_p + F_p)$  
$R$: Recall, defined as $T_p/(T_p + F_n)$  
$Fm$: F-Measure, defined as $(2 * P * R)/(P + R)$

Table 4: Execution times

<table>
<thead>
<tr>
<th>Project</th>
<th>CD-Form Execution time (s)</th>
<th>NiCad Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>netbeans-javadoc</td>
<td>42</td>
<td>8</td>
</tr>
<tr>
<td>eclipse-ant</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>307</td>
<td>75</td>
</tr>
<tr>
<td>j2sdk1.4.0-jdk-swing</td>
<td>276</td>
<td>44</td>
</tr>
</tbody>
</table>

One of the main factors for the performance boost of CD-Form, as compared to the previous prototype tool, is the adoption of the heuristics described in Section 3.4, which make it possible to reduce substantially the number of invocations of the equivalence checker. Table 5 shows the number of comparisons saved thanks to each heuristic on the largest project, j2sdk1.4.0-jdk-swing. The result is that the joint use of all heuristics makes it possible to invoke Grease only 12680 times, i.e., carrying out only the 0.01% of the total theoretical number of comparisons to be performed.

**INIZIO PEZZO NUOVO**

In the previous experiments, the minimum clone size was set to 5 pretty-printed lines. This has sometimes been considered a granularity too small for method clones, especially in Java. So, we have also performed tests with a threshold of 15 pretty-printed lines, using the same procedure as before (for CD-Form, the intermediate step was to use a threshold of 15 bytecode instructions, successively filtering all that corresponds to less than 15 pretty-printed source lines). Table 6 and Table 7 show the results obtained for the above four systems.

**FINE PEZZO NUOVO**

In addition to evaluating CD-Form in terms of clone quality using NiCad as oracle, we have evaluated CD-Form also in terms of its scalability. For this study, four open source systems of increasing size have been analyzed. All these systems are written in Java and have a sufficiently long development history, spanning through many years. The four selected systems are:
Table 5: Sample of the comparisons saved thanks to the use of heuristics

<table>
<thead>
<tr>
<th>Project</th>
<th>j2sdk1.4.0-jdk-swing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of methods</td>
<td>14056</td>
</tr>
<tr>
<td>Number of theoretical comparisons</td>
<td>98778540</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heuristic</th>
<th># of comparisons saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>32475170 (32.88%)</td>
</tr>
<tr>
<td>Number of actions</td>
<td>4053605 (4.10 %)</td>
</tr>
<tr>
<td>Number of distinct actions</td>
<td>61436730 (62.20 %)</td>
</tr>
<tr>
<td>Number of parameters</td>
<td>316939 (0.32 %)</td>
</tr>
<tr>
<td>Transitivity</td>
<td>249275 (0.25 %)</td>
</tr>
<tr>
<td>Different sort</td>
<td>234141 (0.24 %)</td>
</tr>
<tr>
<td>Actual Grease invocations</td>
<td>12680 (0.01 %)</td>
</tr>
</tbody>
</table>

Table 6: Experimental clone detection results (minimum size of 15 bytecode instructions)

<table>
<thead>
<tr>
<th>Project</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>P</th>
<th>R</th>
<th>Fm</th>
</tr>
</thead>
<tbody>
<tr>
<td>netbeans-javadoc</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>eclipse-ant</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>405</td>
<td>8</td>
<td>198</td>
<td>67.16</td>
<td>98.06</td>
<td>79.72</td>
</tr>
<tr>
<td>j2sdk1.4.0-jdk-swing</td>
<td>96</td>
<td>3</td>
<td>1</td>
<td>98.97</td>
<td>96.97</td>
<td>97.96</td>
</tr>
</tbody>
</table>

TP: Clones found by both detectors
FN: Clones found only by NiCad
FP: Clones found only by CD-Form
P: Precision, defined as $T_p/(T_p + F_p)$
R: Recall, defined as $T_p/(T_p + F_n)$
Fm: F-Measure, defined as $(2 \times P \times R)/(P + R)$

Table 7: Execution times (minimum size of 15 bytecode instructions)

<table>
<thead>
<tr>
<th>Project</th>
<th>CD-Form Execution time (s)</th>
<th>NiCad Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>netbeans-javadoc</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>eclipse-ant</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>j2sdk1.4.0-jdk-swing</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
1. ArgoUML, a UML modeling tool that includes support for standard UML diagrams. It is available for download at http://argouml.tigris.org. ArgoUML is made of 195363 Java LOC. This study has considered release 0.34.

2. Apache River, a network architecture for the construction of distributed systems in the form of modular co-operating services. It is available for download at http://river.apache.org and contains 289015 Java LOC. This study has used release 2.2.0.

3. Apache Lucene, a free/open source information retrieval software library. It is available for download at http://lucene.apache.org, and consists of 423139 Java LOC. This study has used release 4.3.0.

4. JBoss, a J2EE compliant application server written in Java. It is available for download at http://www.jboss.org, and contains 581646 Java LOC. This study has used release 6.1.

Table 8 shows the running time (measured in minutes) of CD-Form for the four systems. The experiment of Table 8 also examines how the number of detected clones varies with system size. For the experiments, a minimum size of 15 bytecode instructions in a method was imposed. CD-Form analyzes the first three systems in less than 6 minutes and the last one (about 600 KLOC) in 26 minutes. We think that this is a reasonable response time for a prototype tool, considering the dimension of the systems examined.

The analysis of systems of millions of LOC is too expensive for our tool, and definitely out of the scope of our research, aimed primarily at the use of clone detection for software evolution and maintenance, and not for code provenance analysis.

<table>
<thead>
<tr>
<th>Project</th>
<th>CD-Form Execution time (min.)</th>
<th>Number of detected clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML (~195KLOC)</td>
<td>3</td>
<td>12097</td>
</tr>
<tr>
<td>River (~290KLOC)</td>
<td>1</td>
<td>1988</td>
</tr>
<tr>
<td>Lucene (~423KLOC)</td>
<td>6</td>
<td>2841</td>
</tr>
<tr>
<td>Jboss (~582KLOC)</td>
<td>26</td>
<td>27065</td>
</tr>
</tbody>
</table>

6. Related Work

Many clone detection approaches have been proposed in the literature. For a review of techniques and related tools, see [1]. Depending on the level of analysis applied to the source code, a rough classification recognizes:
• **text-based approaches:** the target source program is considered as sequence of lines/strings. Two code fragments are compared with each other to find longest common subsequences of same text/strings.

• **token-based approaches:** the entire source system is transformed into a sequence of tokens, which is scanned for finding duplicated subsequences.

• **tree-based approaches:** a language-specific parse tree or an abstract syntax tree (AST) is derived from the program. Tree matching is applied to detect similar subtrees.

• **PDG-based approaches:** source code is mapped to a Program Dependency Graph (PDG), on which an isomorphic subgraph matching algorithm is applied for finding similar subgraphs.

• **metrics-based approaches:** metrics are gathered for code fragments; metric vectors are compared instead of comparing code directly.

There are several differences between our proposal and the approaches discussed above. The two distinctive features of our methodology are the use of formal methods and the detection on Java bytecode and not on the source code. Finding similar fragments in binary code and intermediate languages, such as Java bytecode, has not been a major research focus in the clone detection community. Previous work by Baker and Manber [6] has pioneered the clone analysis of Java bytecode. They adopted three tools (Sift, Dup and Diff) designed to find similarity in both source code and text in order to work with bytecode files. Recently, an attempt of performing clone detection on assembler code has been made by Davis and Godfrey in [7]. They use a combination of hill climbing and greedy algorithms to detect clone pairs. In [24] a clone detection technique has been proposed that transforms the source code to an intermediate representation, and then reuses established source-based clone detection techniques to detect clone pairs in the intermediate representation. The authors use as intermediate code the Jimple language which is an alternative representation to the stack-based Java binary code. In [2] the authors present SeByte, a bytecode clone detection approach, which exploits the benefits of compilers (the bytecode representation) for detecting semantically similar methods in Java bytecode. In light of the above, the detection of clones on intermediate code that we chose for CD-Form is not a completely new approach.

To the best of our knowledge, the use of CCS as internal representation of the code and of weak equivalence algorithms to identify clones has never been reported in the literature, and so this is the main contribution of our work. A significant advantage linked to the use of formal methods is that the CCS model generated from a Java program can also be used, using model checking, to perform an automatic analysis aimed at detecting programming errors [25]. In this case, the basic idea is to explore exhaustively the reachable program states, to determine whether a correctness property, expressed in a temporal logic, holds or not. Moreover, model checking, besides proving that a program is not correct, can also help debugging, making it possible to locate errors.
In fact, if a property does not hold, the model checking algorithm generates a counter-example, i.e., an execution trace leading to a state in which the property is violated. This ability to generate counter-examples, which can be exploited to pinpoint the cause of an error, is the main advantage of model checking, as compared to other well-known techniques for software verification, as abstract static analysis.

In light of all the above, the use of formal methods makes it possible to perform clone detection and code analysis via model checking in a single framework, and to get also insight on liveness and safety properties of the code. Intuitively, safety properties express the unreachability of bad states, such as those in which an assertion violation or null pointer dereference has occurred. Liveness properties express that something good eventually happens, e.g., requests are eventually served.

7. Conclusion and Future Work

In this paper the CD-Form tool has been presented, along with the underlying technology used for clone detection. The major contribution of our work is the use of formal methods, which, to the best of our knowledge, have never been used before for detecting code similarities. The experimentation results turning out from the tool validations performed prove the substantial validity of this novel approach, as satisfactory values (over 90%) are obtained both for precision and recall. The main drawback of the current implementation of CD-Form is that it returns only clones of methods. On the plus side, this fixed granularity is good for architectural refactoring.

As compared to the prototype tool used to test our technique, described in [9], CD-Form has been implemented trying to tackle the existing performance problems. The results obtained are fairly satisfactory, as now the tool can analyze projects of hundreds of KLOC in a few minutes. Furthermore, we have corrected a number of accuracy problems linked to the analysis of the bytecode instead of the Java source.

In our opinion, the work done is valuable also in different contexts, apart from clone detection, in that the CCS translation of Java code can be reused to develop other types of code analyzers. As a future work, we plan to extend our technique, making it possible to handle also multiple granularities or to perform general re-engineering. Extensions to detect Type-3 and Type-4 clones by means of more complex and advanced equivalences and model checking are also possible.

References


