A novel approach based on formal methods for clone detection

Antonio Cuomo, Antonella Santone, Umberto Villano
Dept. of Engineering, University of Sannio, Benevento, Italy
cuomoant@unisannio.it, santone@unisannio.it, villano@unisannio.it

Abstract—This paper presents an approach based on formal methods for detecting code clones. The methodology followed performs the analysis on Java bytecode, which is transformed into CCS (Calculus of Communicating Systems) processes which are successively checked for equivalence. A prototype tool targeted at the detection of Type 2 clones is presented. The experiments conducted on programs of different size assess the validity of the proposed approach, pointing out possible improvements for future research.

Keywords—Clone detection; Formal Methods; CCS

I. 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 compiled code is much more difficult than detecting them in source code, because a small change in the source can produce completely different binaries. However, since Java bytecode has become the common way to distribute programs and to execute them through the web, we think that this additional complication is worth the effort.

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 by operational semantics their evolution. 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 [2] 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”.

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 [3] can be used also for verifying safety and liveness program properties. Properties are expressed using temporal logic.

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 [4] 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 [5].

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. The main contribution of this paper is the addition to the description
of the clone detection methodology, preliminarily presented in [6], of actual results obtained through a prototype tool that has been recently implemented. As will be shown in the following, our experiments on programs of different size assess the validity of the proposed approach and look promising for further refinements of the method.

The remainder of the paper is organized as follows: Section II is a review of the basic concepts of CCS and background terms of clone detection, while Section III describes our methodology. In Section IV the prototype tool implementing our approach is briefly presented, and the experimental results we obtained by using it are reported. Finally comparisons with related works are discussed in Section V and our conclusion are presented in Section VI.

II. PRELIMINARIES

In this section, after introducing the basic definitions of code clones, we present the Calculus of Communicating Systems (CCS) [2], the process algebra we have adopted for our clone detection prototype.

A. Clone detection

Software clone detection is an active field of research. In the following a basic introduction to clone detection terminology is given.

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.

B. 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) [2] 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 [2] for further details. The syntax of processes is the following:

\[
p ::= nil \mid x \mid \alpha.p \mid p+p \mid p[p] \mid p\backslash L \mid p[f]
\]

where \( \alpha \) ranges over a finite set of actions \( \mathcal{A} = \{\tau, a, \pi, b, \mu, \ldots\} \). The action \( \tau \in \mathcal{A} \) is called the **internal action**. The set of visible actions, \( \mathcal{V} \), ranged over by \( l, l' \ldots \), is defined as \( \mathcal{A} - \{\tau\} \). Each action \( l \in \mathcal{V} \) (resp. \( l' \in \mathcal{V} \)) has a **complementary action** \( l' \) (resp. \( l \)). The restriction set \( L \), in the processes of the form \( p\backslash L \), is a set of actions such that \( L \subseteq \mathcal{V} \). 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** [2] is given by a relation \( \rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P} \), \( \rightarrow \) is the least relation defined by the rules in Table I.

We now informally explain the semantics for CCS by induction over the structure of processes. \( \text{nil} \) represents a process that can do nothing. There is no rule for \( \text{nil} \) since it cannot evolve. The process \( a.p \) can perform the action \( a \) and thereby become the process \( p \) (rule **Act**). The process \( p+q \) is a process that non-deterministically behaves either as \( p \) or as \( q \) (rule **Sum**). 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 \( q \) (rule **Par**). Furthermore, \( p \) and \( q \) may also together engage in a communication whenever they are able to perform complementary actions. That is, if \( p \) can perform a visible action \( l \) and become \( p' \), and \( q \) can perform \( l \) and become \( q' \), then \( p[q] \) can perform \( \tau \) and become \( p'[q'] \) (rule **Com**). 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 (rule **Res**). 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 \( p'[f] \) (rule **Rel**). The behavior of the process \( x \overset{\text{def}}{=} p \) is that of its definition \( p \) (rule **Con**).

A (labeled) transition system is a quadruple \( T = \)
One of the most popular environments for verifying concurrent systems is the Concurrency Workbench of New Century (CWB-NC) [3], which supports several different specification languages, among which CCS. In the CWB-NC the verification of temporal logic formulae is based on model checking [7], a formal technique for proving the correctness of a system with respect to a desired behavior. This is accomplished by checking whether a structure representing the system (typically a labeled transition system) satisfies a temporal logic formula describing the expected behavior. Moreover, CWB-NC (as all existing verification environments) offers tools to check the above equivalences and to minimize transition systems with respect to them.

III. THE METHODOLOGY

In this section we present the methodology underlying our clone detection approach. 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.

Performing clone detection on Java 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 [8], for real applications there exist actually some differences in the generated bytecode, the amount of which 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: (i) independence of the source programming language (i.e., minor changes to the Java syntax become insignificant); (ii) clone detection without compilation even when source code is lacking; (iii) straight clone detection of merely similar constructs that may “look different” when expressed as source code, avoiding source normalization; (iv) ease of parsing a lower-lever code.

The goal of our methodology and of the tools that have been developed is to capture Type 1 and Type 2 clones at method (function) granularity. It is worth pointing out that Type 1 clones can be detected fairly easily. On the other hand, Type 2 clone detection requires a more substantial effort, since all type information must be ignored, together with variations in identifiers and literals. Our approach requires a three-step process, as almost any other detection technique presented in the literature (albeit with significant

<table>
<thead>
<tr>
<th>Table I STANDARD OPERATIONAL SEMANTICS OF CCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act</td>
</tr>
<tr>
<td>Sum</td>
</tr>
<tr>
<td>Rel</td>
</tr>
<tr>
<td>Par</td>
</tr>
<tr>
<td>Con</td>
</tr>
<tr>
<td>Res</td>
</tr>
</tbody>
</table>

$(S, \mathcal{A}, \rightarrow, p)$, where $S$ is a set of states, $\mathcal{A}$ is a set of transition labels (actions), $p \in S$ is the initial state, and $\rightarrow \subseteq S \times \mathcal{A} \times S$ is the transition relation. If $(p, \alpha, q) \in \rightarrow$, we write $p \xrightarrow{\alpha} q$. If $\delta \in \mathcal{A}^*$ and $\delta = \alpha_1 \ldots \alpha_n, n \geq 1$, we write $p \xrightarrow{\delta} q$ to mean $p \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} q$. Moreover $p \xrightarrow{\lambda} p$, where $\lambda$ is the empty sequence. Given $p \in S$, with $R(p) = \{ q \mid p \xrightarrow{\alpha} q \}$ we denote the set of the states reachable from $p$ by $\rightarrow$. Given a CCS process $p$, the standard transition system for $p$ is defined as $S(p) = (R(p), \mathcal{A}, \rightarrow, p)$.

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. To define the weak equivalence, we 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: $p \xrightarrow{\alpha} p' \xrightarrow{\beta} q' \xrightarrow{\gamma} q$. For each action $\alpha$, we use $\tilde{\alpha}$ 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.

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, $\mathcal{B}$, is a binary relation on $\mathcal{P} \times \mathcal{P}$ such that $p \mathcal{B} q$ implies:
  - (i) $p \xrightarrow{\alpha} r'$ implies $q \xrightarrow{\tilde{\alpha}} q'$ with $p' \mathcal{B} q'$; and
  - (ii) $q \xrightarrow{\alpha} q'$ implies $p \xrightarrow{\alpha} p'$ with $p' \mathcal{B} q'$
- $p$ and $q$ are weak equivalent ($p \approx q$) iff there exists a weak bisimulation $\mathcal{B}$ containing the pair $(p, q)$. 
differences in the intermediate representations used). The three steps are:

- Step 1: transformation of the code into an internal representation;
- Step 2: detection of parts that denote clone pairs;
- Step 3: aggregation of clone pairs into clone classes.

The step 2 is the most time-consuming, whereas the internal representation produced at step 1 has the greatest impact on the accuracy of the results. In the following, the three steps are described in more detail.

We use as internal representation the CCS language. Thus, in the first step, 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 process \( x_j \).

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

\[ T(i) = x_i \overset{\text{def}}{=} \text{storex}.x_{i+1} \]

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

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

\[ T(i) = x_i \overset{\text{def}}{=} \text{loadx}.\text{pushk}.\text{add}.\text{storex}.x_{i+1} \]

\( \text{iinc} \) 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 \( \text{load push add store} \) 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 \( \text{iinc} \) 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 \textit{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 \((\text{ifeq, ifne, } \ldots)\) and, depending on the result, branches or proceeds in sequence. Another set \((\text{if icmpeq, if icmpne, } \ldots)\) 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 \((\text{i.e., both for comparisons of an integer against zero and comparisons of two integers})\) the CCS translation is the same, as follows:

\[ T(i) = x_i \overset{\text{def}}{=} \text{if\_cond}_t.x_{i+1} + \text{if\_cond}_t.x_j \]

The true (resp. false) condition is represented by the CCS action \( \text{if\_cond}_t \) (resp. \( \text{if\_cond}_f \)), while

\[ \text{cond} \in \{\text{eq, ne, icmpeq, icmpne, } \ldots\} \]

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

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

(1)

is translated into Java bytecode using only \text{if\_icmpge} when \( a \) and \( b \) are integer, while it is translated using the pair \text{fcmpq, 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\_icmpge} to that of the pair \text{fcmpq, ifge}. Thus, when transforming in CCS a float branching on condition opcode, we relabel the action representing the opcode \text{fcmpq} to \( \text{tau} \) and we make similar the CCS translation of \text{if\_icmpge} and \text{if\_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:

\[ x_i \overset{\text{def}}{=} \text{if\_icmpge}_t.x_1 + \text{if\_icmpge}_t.x_2 \]

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

\[ x_i \overset{\text{def}}{=} \tau.(\text{if\_ge}_t.x_1 + \text{if\_ge}_t.x_2)[f] \]
where \( f = [\text{if\_icmpge}_\mathit{tt}/\text{if\_ge}_\mathit{tt}, \text{if\_icmpge}_\mathit{tt}/\text{if\_ge}_\mathit{tt}] \)

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

\[
T(i) = x_i \xrightarrow{t} (\text{if\_cond}_\mathit{tt}.x_{i+1} + \text{if\_cond}_\mathit{tt}.x_j)[f]
\]

where \( f = [\text{if\_icmpcond}_\mathit{tt}/\text{if\_cond}_\mathit{tt}, \text{if\_icmpcond}_\mathit{tt}/\text{if\_cond}_\mathit{tt}] \).

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

The second step aims at discovering clones (represented in CCS). 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. Although CCS has been historically used for modeling concurrent systems, in this paper we exploit its algebraic structures mainly for equivalence checking. To check weak equivalence we use the Concurrency Workbench of the New Century (CWB-NC) [3], one of the most popular formal verification tools.

The current version of our tool returns only clones of methods. This implies that \( O(n^2) \) equivalence checks are necessary for \( n \) methods. This complexity can be reduced avoiding any comparison between methods that are clearly different. In practice, only “reasonable” clone candidates are checked. The metric used to guide the choice of the methods to be checked is the number of CCS actions in each CCS process, i.e., in the translation of each method. In the case of Type 2 clone detection, this is a reasonable choice, since Type 2 clones form an equivalence relation, unlike Type 3 clones, where statements can be added or removed.

The previous step returns only clone pairs as result. In order to reduce the amount of output data and improve their readability, a third optional step aggregates clone pairs into classes of mutually equivalent fragments.

IV. REALIZATION AND RESULTS

The methodology presented in the previous section has been used to realize a prototype tool, whose architecture is shown in Figure 1. 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 translated into textual file representations of CCS processes following the methodology described in the previous section. The tool then invokes the CCS equivalence checker, which in the current implementation is the Concurrency Workbench of the New Century (CWB-NC). 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, the tool outputs all the retrieved clone pairs and classes.

As a first validation of the proposed approach, 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 and verified that the detector was 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 necessary explanations in [1]. The detector actually achieved 100% precision and recall in recognizing all the Type 1/Type 2 variations, and none of the Type 3/Type 4 variations (this, until the tool will be expanded to cover these classes, is highly desirable). Obviously, this method alone is not sufficient for a complete validation.

To test the current implementation, we set up a comparison with a mature clone detection tool, NiCad [9]. As the authors say, NiCad is a hybrid parser-based / text comparison clone detection system. NiCad employs a two (plus an optional one)-step process to find clones. In the first phase, a language-sensitive parser extracts functions or blocks of code. These potential clones are then compared with a longest common subsequence (LCS)-based diff algorithm. An optional intermediate step applies code normalizations, in which the potential clones are processed for renaming (blindly or consistently substituting identifiers), filtering or abstraction. These last two techniques allow respectively to discard or abstract in the comparison a subset of the nonterminals of the language-specific grammar employed by the parser. As has been said, our prototype currently is targeted at Type 2 clones. Following the definition of Type 2 clone given in section II, NiCad can be configured to recognize all these clones if one allows for:

- extraction of potential clones at the function level;
- variations in whitespaces, layout and comments: this is always included in NiCad, as in the first step the parser pretty-prints the lines and strips the comments;
- variations in identifiers: this can be configured with NiCad blind or consistent renaming. According to the more general interpretation of the definition, we opt for blind renaming.
- variations in types: obtained by configuring NiCad 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 imposed if NiCad is configured with a threshold of 0.0.

To compare the results of NiCad with our tool, we chose to compare clone pairs. As NiCad 2.9 is only able to output clone classes, where each class contains all the fragments which are mutually clones, we transformed the output to

\(^1\)http://commons.apache.org/bcel/
generate the clone pairs (as we write, the 3.0 release of NiCad has been made available, which directly outputs clone pairs). To match fragments in source code and in bytecode, we leveraged the debug information optionally included in bytecode to obtain the source line number of the first instruction of a method. To consider two fragments equal, it is sufficient that the source line number reported by our tool falls inside the start and end line reported by NiCad. Another problem with the comparison is the granularity mismatch. Both NiCad and our tool support the specification of the granularity. However, while NiCad refers to number of (pretty-printed) source lines, we refer to number of bytecode instructions. To avoid difficulties in the validation, without compromising precision, we decided to choose an unambiguous granularity of 1 for both tools. This leads to a high number of clones, but preserves correctness.

Experiments were conducted on two Java projects. The first one is a social network developed by a student of our University as a final term project in a software engineering lab course. The code is made of 3k lines of Java and, as is typical of projects developed by students, was expected to contain a good amount of clones. The second project is an open-source traffic simulator publicly available on the Internet, VIS-SIM, whose Java core is made of 8k lines of code. The results are reported in Table II, where NiCad is taken as a reference. This means that 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 execution time for each project was about 10 minutes. This is quite high, considering the small scale of the projects: anyway, we found that it is mostly due to the overhead needed to invoke the external command-line CWB-NC from the Java toolchain. We plan to integrate a CCS equivalence checker in the architecture, removing this overhead.

To evaluate the effectiveness of our method, we have to consider both complexity and scalability. From the complexity point of view, it is easy to show that the complexity of the first step (i.e., the extraction of the CCS model from Java bytecode) is linear in the length of the program. In the second step, equivalence checking is used, that is, if $p$ and $q$ are the two CCS processes of two Java bytecode fragments, we have to evaluate the complexity for deciding whether $p$ and $q$ are weak equivalent. In the literature, efficient algorithms exist for computing equivalences: among them there is the algorithm of Paige-Tarjan [10], which is used in the CWB-NC. In the worst case, the complexity in space is $O(n + m)$, where $n$ (resp. $m$) is the size of the transition system for $p$ (resp. $q$).

Scalability is a relevant issue with formal methods, as model and equivalence checking do not scale up well as the size of the system representation grows. However, recently several solutions to scale up formal methods and thus also equivalence checking have been proposed, including abstraction [11] and Binary Decision Diagrams (BDD) [12]. In particular, our approach can be integrated by applying both the abstraction technique based on the selective mu-calculus [13] and the directed model checking [14] with the aim of using heuristic search strategies to improve the efficiency of equivalence checking.

V. 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:

1) 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.

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

3) 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.

4) 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.

5) 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. While detecting
clones on assembler code has been already made (see [4], [5]), the use of both CCS as internal representation of the code and of weak equivalence as algorithm to identify clones are the main innovation in our approach. The advantages of our approach that combines formal methods with the strategy of performing clone detection on Java bytecode are:

1) Irrelevance of formatting: Type I clones are automatically detected since Java bytecode already removes variations in whitespace and comments. Additional source normalizations are not necessary to remove superficial differences such as changes in statement bracketing (i.e., `if (a) b=2;` vs. `if (a) {b=2;}`). 2) The method can be easily extended to detect also Type 3 and Type 4 clones. To manage Type 3 clones, other equivalences can be checked, such as the $\rho$-equivalence proposed in [13]. To deal with Type 4 clones, model checking can be applied: it is sufficient to express the kind of similarity that one wants to verify by means of a logic formula. 3) The CCS model generated by the Java program can also be used to perform code analysis.

### VI. Conclusion and Future Work

In this paper a methodology and a corresponding prototype tool for clone detection are presented. The novel contribution of our work is the use of formal methods, which, to the Authors’ knowledge, have never been used before for detecting code similarities. The presented experimentation proves the substantial validity of the approach, obtaining satisfactory values (over 90%) for both precision and recall. In our opinion the work done is valuable apart from clone detection, in that the CCS translation of Java code can be reused to develop more general static code analyzers. The main drawback of the current implementation is that it returns only clones of methods. On the plus side, this fixed granularity is good for architectural refactoring. As a future work, we intend to extend our technique, making it possible to handle also multiple granularities or general re-engineering. Extensions to detect type III and IV clones by means of advanced equivalence and model checking are also possible. Moreover, our current prototype can be the basis of an optimized tool. At the moment, its main bottleneck is the creation of CCS processes as files that feed an external tool, an issue that can be eliminated by an integrated equivalence checker.

**Acknowledgements**

The Authors thank Domenico Martino for helping in the experimentation and the anonymous reviewers for their valuable comments and suggestions.

### References


