Comparison and Evaluation of Clone Detection Tools

Stefan Bellon, Rainer Koschke, *Member*, *IEEE Computer Society*, Giuliano Antoniol, *Member*, *IEEE*, Jens Krinke, *Member*, *IEEE Computer Society*, and Ettore Merlo, *Member*, *IEEE*

Abstract—Many techniques for detecting duplicated source code (software clones) have been proposed in the past. However, it is not yet clear how these techniques compare in terms of recall and precision as well as space and time requirements. This paper presents an experiment that evaluates six clone detectors based on eight large C and Java programs (altogether almost 850 KLOC). Their clone candidates were evaluated by one of the authors as an independent third party. The selected techniques cover the whole spectrum of the state-of-the-art in clone detection. The techniques work on text, lexical and syntactic information, software metrics, and program dependency graphs.

Index Terms—Redundant code, duplicated code, software clones.

1 Introduction

Reuse through copying and pasting source code is common practice. So-called *software clones* are the results. Sometimes these clones are modified slightly to adapt them to their new environment or purpose. Several authors report 7 percent to 23 percent code duplication [1], [2], [3]; in one extreme case, authors reported 59 percent [4].

The problem with code cloning is that errors in the original must be fixed in every copy. Other kinds of maintenance changes, for instance, extensions or adaptations, must be applied multiple times, too. Yet, it is usually not documented where code was copied. In such cases, one needs to detect them. For large systems, detection is feasible only by automatic techniques. Consequently, several techniques have been proposed to detect clones automatically [1], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. The abundance of techniques calls for quantitative evaluations.

This paper presents an experiment conducted in 2002 that evaluates six clone detectors based on eight large C and Java programs (altogether almost 850 KLOC). The experiment involved several researchers who applied their tools on these systems. Their clone candidates were evaluated by one of the authors, namely, Stefan Bellon, as an independent third party. The selected techniques cover the whole spectrum of the state of the art in clone detection. The techniques work on text, lexical and syntactic information, software metrics, and program dependency graphs. Fig. 1 lists the participants, their tools, and the type of information they leverage.

The remainder of this paper is organized as follows: The next section describes the techniques we evaluated and related techniques for clone detection. Section 3 gives an operational structural definition of clone types used in the evaluation. The setup for the experiment is described in Section 4 and its results are presented in Section 5. Section 6 describes related research in clone detection evaluation.

- *S. Bellon is with Axivion GmbH, Nobelstr. 15, 70569 Stuttgart, Germany. E-mail: bellon@axivion.com.*
- *R. Koschke is with the Universität Bremen, Fachbereich 03, Postfach 33 04 40, 28334 Bremen, Germany. E-mail: koschke@tzi.de.*
- *G. Antoniol is with the Département de Génie Informatique, École Polytechnique de Montréal, Pavillons Lassonde, MacKay-Lassonde, 2500, chemin de Polytechnique, Montréal (Quebec), Canada, H3T 1J4. E-mail: antoniol@ieee.org.*
- *J. Krinke is with Fern-Universität in Hagen, Universitätsstr. 27, 58097 Hagen, Germany. E-mail: krinke@ieee.org.*
- *E. Merlo is with the Department of Computer Engineering, Ecole Polytechnique of Montreal, PO Box 6079, Station Downtown, Montreal (Quebec), Canada, H3C 3A7. E-mail: ettore.merlo@polymtl.ca.*

Manuscript received 11 Apr. 2006; revised 21 Oct. 2006; accepted 14 May 2007; published online 10 July 2007.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0089-0406.
Digital Object Identifier no. 10.1109/TSE.2007.70725.

2 Clone Detection

Software clone detection is an active field of research. This section summarizes research in clone detection.

Textual comparison. The approach of Ducasse et al. compares whole lines to each other textually [4]. To increase performance, lines are partitioned using a hash function for strings. Only lines in the same partition are compared. The result is visualized as a dot plot, where each dot indicates a pair of cloned lines. Clones may be found as certain patterns in those dot plots visually. Consecutive lines can be summarized to larger cloned sequences automatically as uninterrupted diagonals or displaced diagonals in the dot plot.

Johnson [13] uses the efficient string matching by Karp and Rabin [14] based on fingerprints.

Token comparison. Baker's technique is also a line-based comparison. Instead of a string comparison, the token sequences of lines are compared efficiently through a suffix tree. First, each token sequence for a whole line is summarized by a so-called *functor* that abstracts from concrete values of identifiers and literals [1]. The functor characterizes this token sequence uniquely. Assigning functors can be viewed as a perfect hash function. Concrete values of identifiers and literals are captured as parameters

Participant	Tool	Comparison
Brenda S. Baker	`Dup`	Token
Ira D. Baxter	`CloneDR`	AST
Toshihiro Kamiya	`CCFinder`	Token
Jens Krinke	`Duplix`	PDG
Ettore Merlo	`CLAN`	Function Metrics
Matthias Rieger	`Duploc`	Text

Fig. 1. Participating scientists. CloneDR is a trademark of Semantic Designs Inc.

to this functor. An encoding of these parameters abstracts from their concrete values but not from their order so that code fragments may be detected that differ only in systematic renaming of parameters. Two lines are clones if they match in their functors and parameter encoding.

The functors and their parameters are summarized in a suffix tree, a trie that represents all suffixes of the program in a compact fashion. A suffix tree can be built in time and space linear to the input length [7], [15]. Every branch in the suffix tree represents program suffixes with common beginnings, hence, cloned sequences.

Kamiya et al. increase recall for superficially different yet equivalent sequences by normalizing the token sequences [9].

Because syntax is not taken into account, the found clones may overlap different syntactic units, which cannot be replaced through functional abstraction. In either a preprocessing [16], [17] or a postprocessing [18] step, clones that completely fall in syntactic blocks can be found if block delimiters are known.

Metric comparison. Merlo et al. gather different metrics for code fragments and compare these metric vectors instead of comparing code directly [2], [3], [12], [19]. An allowable distance (for instance, euclidean distance) for these metric vectors can be used as a hint for similar code. Specific metric-based techniques were also proposed for clones in Web sites [20], [21].

Comparison of abstract syntax trees (AST). Baxter et al. partition subtrees of the abstract syntax tree of a program based on a hash function and then compare subtrees in the same partition through tree matching (allowing for some divergences) [8]. A similar approach was proposed earlier by Yang [22] using dynamic programming to find differences between two versions of the same file.

Comparison of program dependency graphs (PDG). Control and data flow dependencies of a function may be represented by a program dependency graph; clones may be identified as isomorphic subgraphs [10], [11]; because this problem is NP-hard, Krinke uses approximative solutions.

Other techniques. Marcus and Maletic use latent semantic indexing (an information retrieval technique) to identify fragments in which similar names occur [23]. Leitao [24] combines syntactic and semantic techniques through a combination of specialized comparison functions that compare various aspects (similar call subgraphs, commutative operators, user-defined equivalences, and transformations into canonical syntactic forms). Each comparison function yields an evidence that is summarized in an evidence-factor model yielding a clone likelihood. Wahler et al. [25] and Li et al. [26] cast the search for similar fragments as a data mining problem. Statement sequences are summarized to item sets. An adapted data mining algorithm searches for frequent item sets.

3 Basic Definitions

This section presents definitions that form the foundation for the evaluation. These definitions represent the consensus among all participants of the experiment accounting for the different backgrounds of the participants.

The foremost question to answer is, "What is a clone?" Roughly speaking, two code fragments form a clone pair if they are similar enough according to a given definition of similarity. Different definitions of *similarity* and associated levels of tolerance allow for different kinds and degrees of clones.

A piece of code, A, is similar to another piece of code, B, if B subsumes the functionality of A; in other words, they have "similar" preconditions and postconditions. We call such a pair (A, B) a *semantic* clone. Unfortunately, detecting semantic clones is undecidable in general.

Another definition of similarity considers the program text: Two code fragments form a clone pair if their program text is similar. The two code fragments may or may not be equivalent semantically. These kinds of clones are often the result of *copy&paste*; that is, the programmer selects a code fragment and copies it to another location.

Copy&paste is a frequent programming practice and an example of ad hoc reuse. The automatic clone detectors evaluated in this experiment find clones that are similar in program text and, hence, the latter definition of a clone pair is adopted in this paper.

Clones of this nature may be compared on the basis of the program text that was copied. We can distinguish the following types of clones:

- **Type 1** is an exact copy without modifications (except for white space and comments).
- **Type 2** is a syntactically identical copy; only variable, type, or function identifiers were changed.
- **Type 3** is a copy with further modifications; statements were changed, added, or removed.

Some of the tools report so-called parameterized clones [6], which are a subset of type-2 clones. Two code fragments A and B are a parameterized clone pair if there is a bijective mapping from A's identifiers onto B's identifiers that allows an identifier substitution in A resulting in A' and A' is a type-1 clone to B (and vice versa).

Differentiating parameterized clones would have required us to check for consistent renaming when we evaluated the clone pairs proposed by the tools. Because the validation was done completely manually and because not all tools make this distinction, we did not distinguish parameterized clones from other type-2 clones.

While type-1 and type-2 clones are precisely defined and form an equivalence relation, the definition of type-3 clones is vague. Some tools consider two consecutive type-1 or type-2 clones together forming a type-3 clone if the gap in between is below a certain threshold of lines. Another precise definition could be based on a threshold for the

Original:

```
while (c1)
{
  if (c2)
  {
    yes();
  }
  else
  {
    no();
  }
}
```

Normalized:

```
while (c1) {
  if (c2) {
    yes(); }
  else {
    no(); } }
```

Fig. 2. Original code and the same code normalized.

Levenshtein Distance, that is, the number of deletions, insertions, or substitutions required to transform one string into another.

Because there is no consensus on a suitable similarity measure for type-3 clones, all clones reported by the evaluated tools that are not type-1 or type-2 clones fall into the category type-3 in our study. It is then the decision of the human analyst whether type-3 clone candidates are real clones.

We are now in a position to define clone pairs more precisely:

Definition 1. *A* clone (pair) *is a triple* (f_1, f_2, t) *where* f_1 *and* f_2 *are two similar code fragments and* t *is the associated type of similarity (type 1, 2, or 3).*

As a matter of fact, in the evaluation, we further constrained the above definition by the additional requirement that clones may be replaced through function calls, that is, that they are syntactically complete. Some of the tools report code fragments that are at different syntactic nesting levels (e.g., a fragment consisting of parts of two different consecutive function bodies), which could indeed be replaced through macros; but a maintenance programmer would never want to replace them because the replacement would make it hard to understand the program.

So, the next question is, "What is a code fragment, exactly?" We could treat a sequence of tokens as a code fragment. Yet, the notion of a token differs from tool to tool (e.g., are preprocessor tokens considered?) and not all tools report token sequences. Rather than tokens, our definition of code fragments is based on text. Tokens may be mapped onto text and the source text is a less debatable point of reference (it is only *less* debatable rather than *not at all* debatable because of macros and preprocessor directives in whose presence one could use the preprocessed or original text).

Program text may be referenced by filename and row and column information. Unfortunately, not all tools report column information. Thus, the least common denominator for the definition of a code fragment for our evaluation is filename and row information.

Definition 2. *A* code fragment *is a tuple* (f, s, e) *which consists of the name of the source file* f*, the start line* s*, and the end line* e *of the fragment. Both line numbers are inclusive.*

Program	Language	Program size
weltab (by Elliot Chikofsky)	C	11K SLOC
cook [27]	C	80K SLOC
snns [28]	C	115K SLOC
postgresql [29]	C	235K SLOC
netbeans-javadoc [30]	Java	19K SLOC
eclipse-ant [31]	Java	35K SLOC
eclipse-jdtcore [31]	Java	148K SLOC
j2sdk1.4.0-javax-swing [32]	Java	204K SLOC

Fig. 3. Overview of the programs used in the main run.

4 Experimental Setup

This section explains how the experiment was set up. Explanations of our general idea as well as in-depth descriptions of the metrics used for the comparison will be given.

4.1 Preparations

We analyzed C and Java systems. Using two different languages and systems of different sizes decreases the degree of bias.

We conducted the experiment in two phases: a test run and the main experiment.

4.1.1 Test Run

The goal of the test run was to identify potential problems for the main run. The test phase analyzed two small C programs (bison and wget) and two small Java programs (EIRC and spule).

In the test run, we noticed that some tools report the start and end lines of the code fragments a line earlier or later if the lines consist of only a brace. In practice, this difference is irrelevant, but it complicates the comparison of clones from different tools.

For this reason, the source code for the main run was "normalized." Empty lines were removed. Lines containing only opening or closing braces were removed and the braces were added to the line above, paying attention to single-line comments, etc. (see Fig. 2).

Tools using layout information [12] in order to detect clones may be affected by this normalization, but to make the comparison easier, all participants agreed to the normalization.

4.1.2 Main Run

The main run consisted of the analysis of four programs written in C and four Java programs. The size of the source code of the programs varied from 11K SLOC to 235K SLOC. Fig. 3 gives an overview of the programs used in the experiment.

As some tools can be configured, we split the main run into a mandatory and a voluntary part. The mandatory part has to be done with the "default" settings of the particular tool, whereas in the voluntary run, each scientist could tune the settings of his or her tool based on her or his own experimentation with the subject system in order to gain the best results.

The tools were operated by the participants in a fixed period of time (five weeks) and the results were collected and evaluated by Stefan Bellon.

By consensus among all participants, only clones that are at least six lines long were reported. Smaller clones tend to be more spurious. Some of the tools applied a preprocessor before they did the analysis; others worked directly on the original program text.

4.2 Benchmark

We compared the individual results from the participants against a reference corpus of "real clones" similarly to the evaluation scheme in information retrieval. Each clone pair suggested by a tool will be called *candidate* and each clone pair of the reference corpus will be called *reference* in the following.

The obvious, naive ways to create such a reference corpus are:

1. union of candidates reported by different tools,
2. intersection of candidates reported by different tools, and
3. candidates that were found jointly by N tools.

All three ways have deficiencies. The first alternative will result in a precision of 1 for each tool as all the candidates a tool reports are present in the reference corpus. Additionally, we get many spurious false positives among the references. The second alternative has the reverse effect: The recall for all tools is 1 and we obtain many spurious true negatives (it suffices that a single tool cannot detect a certain clone). The third alternative is a compromise between the first two and does not really help either. Apart from the fact that we have to justify the chosen value of N, there can always be N tools that report the same false positive, or only $N-1$ tools find a true positive.

Instead, we built the reference corpus manually. Stefan Bellon—as an independent party (referred to as *oracle* in the following—looked at 2 percent of all 325,935 submitted candidates and built a reference corpus by inserting proposed candidates (sometimes after having modified them slightly). In the following, we will use the term *oracled* for all candidates viewed by Stefan Bellon to decide whether or not to accept it as a clone. Please note that *oracled* includes *rejected* and *accepted* as is or in varied form.

An automatic selection process made sure that he did not know which tools proposed the candidate and that the 2 percent was distributed equally, so that no tool is preferred or discriminated against. As much as we wished to classify more than just 2 percent of the candidates, it was impossible considering our time constraints: It took 44 hours to classify the first 1 percent and another 33 hours for the second 1 percent.

We anticipated this problem in the design of the experiment and took two countermeasures. First, one evaluation was done after 1 percent of the candidates had been oracled. Then, another 1 percent was oracled. The interesting observation (as can be seen in Section 5.3) was that the relative quantitative results are almost the same. Second, we injected clones that we did not disclose to the participants in the given programs. The injected clones helped us to get a better idea of the potential recall. Fig. 4 shows how many clone pairs of which clone type were injected into the programs and how many were found by the union of the tools.

Program	injected Type 1	found Type 1	injected Type 2	found Type 2	injected Type 3	found Type 3	Sum of injected	Sum of found
weltab	5	5	6	6	7	5	18	16
cook	1	1	1	1	1	1	3	3
snns	1	1	0	0	1	1	2	2
postgresql	0	0	2	2	0	0	2	2
netbeans-javadoc	4	4	6	5	6	2	16	11
eclipse-ant	0	0	2	2	1	1	3	3
eclipse-jdtcore	2	2	0	0	1	1	3	3
j2sdk1.4.0-javax-swing	0	0	1	1	2	2	3	3
Sum	13	13	18	17	19	13	50	43

Fig. 4. Injected secret clones.

The distribution of the injected clones among the programs is not even as Stefan Bellon started introducing many clones in two programs and then noticed that he would exceed his time constraints. After injecting the clone pairs into the programs, they were added to the reference corpus as well.

4.3 Methods of Evaluation—Metrics

This section defines the measurements taken to compare the automatic clone detection tools.

The evaluation is based on clone pairs rather than equivalence classes of clones because, only for type-1 and type-2 clones, the underlying similarity function is reflexive, symmetric, and transitive. The similarity of type-3 clones is not transitive: If A is a type-3 clone of B and B one of C, the similarity between A and C might be too low to qualify it as type-3 clone. Moreover, some tools report their clones not as classes but as clone pairs.

In order to determine whether a candidate matches a reference, we need a precise measurement. Pragmatically, we did not insist on completely overlapping code fragments but allowed a "sufficiently large" overlap between candidates and reference clone pairs.

Definition 3. Overlap *is the ratio of code common to two code fragments, CF_1 and CF_2, i.e., their intersection correlated to their union. Let* $\text{lines}(CF)$ *denote the set of lines of a code fragment CF; then,* $\text{overlap}(CF_1, CF_2)$ *is defined as:*

$$\text{overlap}(CF_1, CF_2) = \frac{|\text{lines}(CF_1) \cap \text{lines}(CF_2)|}{|\text{lines}(CF_1) \cup \text{lines}(CF_2)|}.$$

Definition 4. Contained *is the ratio of code of one code fragment contained in another one. Let* $\text{lines}(CF_1)$ *denote the set of lines of the first code fragment and* $\text{lines}(CF_2)$ *the set of lines of the second code fragment; then,* $\text{contained}(CF_1, CF_2)$ *is defined as:*

$$\text{contained}(CF_1, CF_2) = \frac{|\text{lines}(CF_1) \cap \text{lines}(CF_2)|}{|\text{lines}(CF_1)|}.$$

Now, we use the above two definitions to create two metrics that tell us how well a candidate hits a reference.

For the following two definitions to work, we have to make sure that the two code fragments CF_1 and CF_2 that make up a clone pair are ordered as follows:

$$\begin{aligned} CF_1 < CF_2 \Leftrightarrow\ & (CF_1.Filename < CF_2.Filename) \vee \\ & (CF_1.Filename = CF_2.Filename \wedge \\ & CF_1.StartLine < CF_2.StartLine) \vee \\ & (CF_1.Filename = CF_2.Filename \wedge \\ & CF_1.StartLine = CF_2.StartLine \wedge \\ & CF_1.EndLine < CF_2.EndLine). \end{aligned}$$

Thus, for a valid clone pair $CP = (CF_1, CF_2, t)$, $CF_1 < CF_2$ must always hold (code fragments of candidates with the wrong order are simply swapped in order to meet this criterion).

Definition 5. *The good-value between two clone pairs CP_1 and CP_2 is defined as follows:*

$$\begin{aligned} \text{good}(CP_1, CP_2) = \min(&\text{overlap}(CP_1.CF_1, CP_2.CF_1), \\ &\text{overlap}(CP_1.CF_2, CP_2.CF_2)). \end{aligned}$$

Two clone pairs CP_1 and CP_2 are thus called a good-match(p) iff, for $p \in [0,1]$, holds

$$\text{good}(CP_1, CP_2) \geq p.$$

We are using the minimum degree of overlap because it is stricter than the maximum or average.

Definition 6. *The ok-value between two clone pairs CP_1 and CP_2 is defined as follows:*

$$\begin{aligned} \text{ok}(CP_1, CP_2) = \min(&\max(\text{contained}(CP_1.CF_1, CP_2.CF_1), \\ &\text{contained}(CP_2.CF_1, CP_1.CF_1)), \\ &\max(\text{contained}(CP_1.CF_2, CP_2.CF_2), \\ &\text{contained}(CP_2.CF_2, CP_1.CF_2))). \end{aligned}$$

Two clone pairs CP_1 and CP_2 are thus called an ok-match(p) iff, for $p \in [0,1]$, holds:

$$\text{ok}(CP_1, CP_2) \geq p.$$

The meanings of the good-value and ok-value can be seen easily by way of an example. An ok-match(p) applies if, in at least one direction, a clone pair is contained in another one for a portion of more than (or equal to) $p \cdot 100\,\%$; that is, one fragment subsumes another one sufficiently. However, this leads to the anomaly that one clone pair can be a lot larger than the other one. With the good-match(p) criterion, this cannot happen as the intersection of both clone pairs is used. The example of Fig. 5 illustrates this.

The vertical line in the middle symbolizes the linear source code. The first source line is at the top; the last one is at the bottom. The code fragments of the participating clone pairs are represented by the filled rectangles. The left side stands for the first clone pair; the right side stands for the second. The dotted arrows symbolize how the code fragments were copied. Let us assume that the left side is the clone candidate and the right side is a clone pair from the reference corpus. The first code fragment of the candidate is one line shorter and starts and ends earlier than the corresponding code fragment of the reference. The second code fragment of the candidate, however, is completely contained within the corresponding code fragment of the reference but two lines shorter.

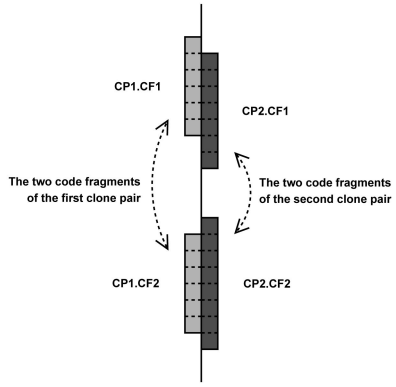

Fig. 5. Example of overlapping of two clone pairs.

This yields a good-value as follows:

$$\text{good}(CP_1, CP_2) = \min\left(\frac{5}{8}, \frac{6}{8}\right) = \frac{5}{8} < 0.7 = p.$$

Thus, the example does not satisfy the criterion for a good-match(0.7).

The ok-value is calculated as:

$$\begin{aligned} \text{ok}(CP_1, CP_2) = \min\Bigg(&\max\left(\frac{5}{6}, \frac{5}{7}\right), \\ &\max\left(\frac{6}{6}, \frac{6}{8}\right)\Bigg) = \frac{5}{6} > 0.7 = p. \end{aligned}$$

Thus, the example is an ok-match(0.7).

The following inequality always holds:

$$\text{ok}(CP_1, CP_2) \geq \text{good}(CP_1, CP_2).$$

The inequality means that a good-match(p) is a stronger criterion than an ok-match(p) for the same value of p. In our experiment, we decided to use a value of $p = 0.7$. Because the threshold for the acceptable length of a clone was 6 in the experiment, the choice of $p = 0.7$ allows two six-line code fragments to be shifted by one line. For instance, if one clone pair's fragment starts at line 1 and ends at 6, and the other's fragment starts at line 2 and ends at 7, the degree of overlap is $5/7 > 0.7 = p$. This choice accommodates the off-by-one disagreement in the line reporting of the evaluated tools. Because both measures are essentially measures of overlaph—-*good* from the perspective of both fragments and *ok* from the perspective of the smaller fragment—we chose to use the same threshold for both measures for reasons of uniformity.

Finally, a mapping from candidates to references has to be established. Each candidate is mapped to the reference that it best matches. The idea of the algorithm for establishing this mapping is shown in Fig. 6 (in reality, a more efficient implementation is used).

There are two dimensions to optimize for the mapping from candidates onto references: the *good* and *ok* values.

```
for r in References loop
  for c in Candidates loop
    ok_max := ok(c, BestReferenceOf[c]);
    good_max := good(c, BestReferenceOf[c]);
    ok := ok(c,r);
    good := good(c,r);
    if better then
      BestReferenceOf[c] := r;
    end if;
  end loop;
end loop;
```

Fig. 6. Mapping algorithm.

Predicate `better` in Fig. 6 is used to define the optimization criterion. The motivation for predicate `better` is to map references to candidates that result in a *good* value as high as possible and in an *ok* value as high as possible without compromising the *good* value (i.e., the match remains in the *good* category). Fig. 7 is used to explain the constituents of `better` in detail. In Fig. 7, you can see the direction of each constituent of `better` by means of the path the (good, ok) coordinate of the best matching reference takes during the algorithm: If possible, it moves towards the upper right corner of the good/ok value space. The shaded part below the diagonal is irrelevant because $good > ok$ always holds. Predicate `better` yields `true` if at least one of the following properties is true:

1. $\texttt{good} \geq p \wedge \texttt{good} > \texttt{good\_max}$. The good-value increases and climbs above threshold p (arrows leading into the small triangle in the top right corner of Fig. 7).
2. $\texttt{good} = \texttt{good\_max} \wedge \texttt{ok} > \texttt{ok\_max}$. The good-value stays the same and the ok-value increases (arrows leading vertically upward in Fig. 7).
3. $\texttt{ok} \geq p \wedge \texttt{ok\_max} < p$. The ok-value climbs from below threshold p to above threshold p (arrows leading from the white triangle into the gray rectangle and not being vertical arrows in Fig. 7).

4.3.1 Quality of the Detection

Based on the above definitions, we can define several measures to determine various aspects of detection quality. In the following definitions, T is a variable denoting one of the participating tools, P is a variable denoting one of the analyzed programs, and τ is a variable denoting the clone type that is observed. All three variables have a special value "all" referring to all tools, programs, and clone types, respectively. τ, furthermore, has a special value "unknown" as some tools cannot categorize clone types.

First, we introduce some base numbers. Then, we define recall and precision using these base numbers.

Definition 7. *Let* $\mathrm{Cands}(P,T,\tau)$ *denote the candidates that are reported for program* P *by tool* T *regarding the clone type* τ.

Let $\mathrm{Refs}(P,\tau)$ *denote the references that are present in the reference corpus for program* P *regarding clone type* τ.

Let $\mathrm{OKRefs}(P,T,\tau)$ *denote references of program* P *and clone type* τ *that are matched with* $\mathrm{ok} \geq 0.7$ *by tool* T.

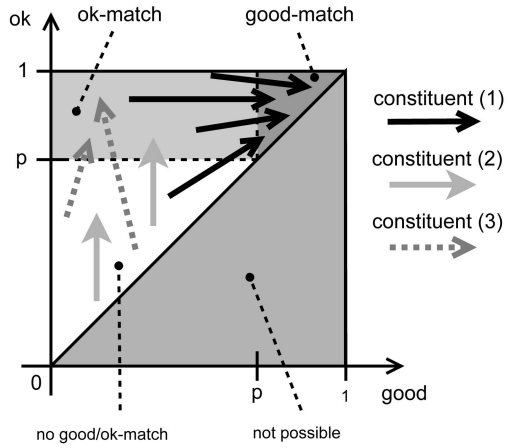

Fig. 7. Graphical representation function `better`.

Let $\mathrm{GoodRefs}(P,T,\tau)$ *denote references of program* P *and clone type* τ *that are matched with* $\mathrm{good} \geq 0.7$ *by tool* T.

Let $\mathrm{DetectedRefs}(P,T,\tau)$ *denote a synonym either for* $\mathrm{OKRefs}(P,\ T,\tau)$ *or for* $\mathrm{GoodRefs}(P,T,\tau)$*, depending on whether it is used in* good *or* ok *context.*

Let $\mathrm{FoundSecrets}(P,T,\tau)$ *denote the number of injected secret clones of clone type* τ *that tool* T *found for program* P.

Let $\mathrm{OracledCands}(P,T,\tau)$ *denote all candidates of tool* T *for program* P *and clone type* τ *that were investigated by the oracle.*

Let $\mathrm{RejectedCands}(P,T,\tau)$ *denote all candidates of tool* T *for program* P *and clone type* τ *that were investigated by the oracle but for which* $\mathrm{good} < 0.7$ *or* $\mathrm{ok} < 0.7$*, respectively (depending on the context), hold after the human investigation.*

Let $\mathrm{TrueNegativeRefs}(P,T,\tau)$ *denote all references of clone type* τ *and program* P *of the reference corpus where no candidate of tool* T *produces* $\mathrm{good} \geq 0.7$ *or* $\mathrm{ok} \geq 0.7$ *(depending on the context) after the evaluation.*

The reference set is formed by clones proposed by at least one tool that was confirmed by the human oracle and by those clones that we injected manually. Consequently, a true negative across all tools exists when an injected clone was not found or the human oracle has modified the clone pair so the resulting reference was not matched by a candidate in any tool's set (even the candidate that caused the clone to be put into the reference set in the first place).

Based on the above base numbers, we can define recall and precision as follows:

Definition 8.

$$\mathrm{Recall}(P,T,\tau) = \frac{|\mathrm{DetectedRefs}(P,T,\tau)|}{|\mathrm{Refs}(P,\tau)|},$$

$$\mathrm{Precision}(P,T,\tau) = \frac{|\mathrm{DetectedRefs}(P,T,\tau)|}{|\mathrm{Cands}(P,T,\tau)|}.$$

The recall metric is meaningful even though we oracled only 2 percent, but the precision metric applied to our incomplete reference set would yield only a lower bound of the actual precision. That is why we use a different metric to

get an idea of the quality of the proposed candidates that is based on the candidates we actually validated as follows:

Definition 9.

$$\text{Rejected}(P,T,\tau) = \frac{|\text{RejectedCands}(P,T,\tau)|}{|\text{OracledCands}(P,T,\tau)|}.$$

Note that Rejected equals $1 - \text{Precision}$ if all candidates are oracled.

4.3.2 Clone Size Aspects

Next, we define aspects of clone sizes measured as lines of code. With these measurements, we can investigate the volume of code proposed as clones by the tools.

Definition 10. *Let* $\text{size}(CF)$ *denote the size of a code fragment* CF *as follows:*

$$\text{size}(CF) = CF.EndLine - CF.StartLine + 1.$$

Definition 11. *Let* $\text{MaxRefSize}(P,\tau)$ *denote the size of the reference of program* P *with clone type* τ *and maximal clone pair size, where the following definition of size for a clone pair* CP *is used:*

$$\text{size}(CP) = \max(\text{size}(CP.CF_1), \text{size}(CP.CF_2)).$$

Let $\text{MaxCandSize}(P,T,\tau)$ be defined analogously to MaxRefSize.

Definition 12. *Let* $\text{AvgRefSize}(P,\tau)$ *denote the average size of the references of program* P *with clone type* τ*, where the following definition of size of a clone pair* CP *is used:*

$$\text{size}(CP) = \frac{\text{size}(CP.CF_1) + \text{size}(CP.CF_2)}{2}.$$

Let $\text{AvgCandSize}(P,T,\tau)$, $\text{StdDevRefSize}(P,\tau)$ (standard deviation), and $\text{StdDevCandSize}(P,T,\tau)$ be analogously defined.

4.3.3 Cloning Scope

The following measurements allow us to investigate whether code is more often copied within files or across files.

Definition 13. *Let* $\text{IntraFileRefs}(P,\tau)$ *denote all references of program* P *and clone type* τ*, where, for each reference* CP*, this equality holds:*

$$CP.CF_1.Filename = CP.CF_2.Filename.$$

Let $\text{IntraFileCands}(P,T,\tau)$ be defined analogously.

Definition 14. *Let* $\text{AcrossFileRefs}(P,\tau)$ *denote all references of program* P *and clone type* τ *where for each reference* CP *this inequality holds:*

$$CP.CF_1.Filename \neq CP.CF_2.Filename.$$

Let $\text{AcrossFileCands}(P,T,\tau)$ be analogously defined.

4.3.4 Distinctiveness of Tools

The measurements defined in this section allow us to investigate the distinctive contribution of a tool and also its distinctive deficiencies.

Definition 15. *Let* $\text{OnlyRefs}(P,T,\tau)$ *denote all references of program* P *and clone type* τ *for which only tool* T *has candidates with* $\text{good} \geq 0.7$ *or* $\text{ok} \geq 0.7$*, respectively (depending on the context).*

Definition 16. *Let* $\text{OnlyButOneRefs}(P,T,\tau)$ *denote all references of program* P *and clone type* τ *for which tool* T *is the only tool that has no candidates with* $\text{good} \geq 0.7$ *or* $\text{ok} \geq 0.7$*, respectively (depending on the context).*

Definition 17. *Let* $\text{OverlappingCands}(P,T,\tau)$ *denote all candidates of program* P*, tool* T *and clone type* τ *for which this inequality for each clone pair* CP *holds:*

$$CP.CF_1.EndLine \geq CP.CF_2.StartLine.$$

5 Evaluation

This section describes the results of the experiment. First, we present the reference set. Then, we take a closer look at one of the analyzed programs. Finally, have a look at the results of a few of the other analyzed programs.

We must note that Merlo et al. used two different clone detection techniques in this experiment: a metric-based one for function clones and a token-based one for type-1 and type-2 clones.

5.1 The Reference Set

Jens Krinke's tool is able to analyze C systems only. All other tools handle both C and Java. Krinke was not able to analyze the largest of the C programs, namely, postgresql. Matthias Rieger was not able to analyze postgresql and the largest Java program, namely, j2sdk.1.4.0-javax-swing. Hence, values for those programs do not exist for those two tools because of scalability issues.

We should also note that according to the experimental setup, clones must consist of contiguous pieces of code, which puts Krinke's tool at a disadvantage. His tool takes only control and data flow into account and is independent of the textual order of statements, so it may report clones that need not consist of consecutive lines of code.

Fig. 8 presents an overview of the number of references and candidates involved. The last three columns reflect the state after 2 percent of the candidates were oracled.

The yield value in Fig. 8 is the percentage of oracled candidates that were accepted as references:

$$|\text{Refs}|/|\text{OracledCands}|.$$

It is a meaningful value for a single tool. To give an idea of the overall acceptance rate, we are using it across tools here. However, across several tools, the value must be taken with caution because two identical oracled and accepted candidates from two tools end up as only one reference. They count as two oracled candidates but only as one accepted reference. Because oracling two identical candidates is negligible given the high absolute number and the low

Program	|Cands|	|OracledCands|	|Refs|	Yield (in %)
weltab	13901	280	252	90.00
cook	27122	544	402	73.90
snns	66331	1329	903	67.95
postgresql	59114	1182	555	46.95
netbeans-javadoc	7860	159	55	34.59
eclipse-ant	2440	51	30	58.82
eclipse-jdtcore	92905	1856	1345	72.47
j2sdk1.4.0-javax-swing	56262	1127	777	68.94
Total	325935	6528	4319	66.16

Fig. 8. Candidates and references after 2 percent of the candidates were oracled.

percentage of candidates we actually looked at in our experiment, the yield value in Fig. 8 is still a meaningful lower bound for the overall acceptance rate.

Figs. 9 and 10 show the influence of our choice for the ok-metric and good-metric. You can easily see that both the number of rejected candidates and the number of true negatives increase when looking at the good-matches compared to the ok-matches, which is the expected behavior. The good-metric is more strict than the ok-metric and, therefore, the misses occur at a higher rate: More candidates are rejected because they do not match the references according to the good-metric and more references are missed because the candidates do not overlap with them well enough.

Please note that RejectedCands is the number of candidates that were considered by the oracle but—in the part of the experiment when all candidates are compared with the reference set—matched no reference. This can happen for three reasons: The reference is an injected hidden clone, or the candidate was rejected by the oracle, or the candidate was changed by the oracle so much that the

Program	|RejectedCands|	$\frac{\|\text{RejectedCands}\|}{\|\text{Cands}\|}$	|TrueNegativeRefs|	$\frac{\|\text{TrueNegativeRefs}\|}{\|\text{Cands}\|}$
weltab	33	11.79	3	1.19
cook	151	27.76	2	0.50
snns	352	26.49	4	0.44
postgresql	685	57.95	31	5.59
netbeans-javadoc	107	67.30	5	9.09
eclipse-ant	23	45.10	0	0.00
eclipse-jdtcore	624	33.62	28	2.08
j2sdk1.4.0-javax-swing	405	35.94	14	1.80

Fig. 9. Rejected candidates and unrecognized references using the ok-match criterion after 2 percent of the candidates were oracled.

Program	|RejectedCands|	$\frac{\|\text{RejectedCands}\|}{\|\text{Cands}\|}$	|TrueNegativeRefs|	$\frac{\|\text{TrueNegativeRefs}\|}{\|\text{Cands}\|}$
weltab	119	42.50	26	10.32
cook	265	48.71	58	14.43
snns	755	56.81	141	15.61
postgresql	886	74.96	183	32.97
netbeans-javadoc	133	83.65	18	32.73
eclipse-ant	28	54.90	2	6.67
eclipse-jdtcore	1167	62.88	395	29.37
j2sdk1.4.0-javax-swing	756	67.08	96	12.36

Fig. 10. Rejected candidates and unrecognized references using the good-match criterion after 2 percent of the candidates were oracled.

original candidate failed to match closely enough (with "closely enough" defined by the ok or the good-metric). This number is reported in column 1 in Figs. 9 and 10. The number of candidates that were rejected outright is the difference between the numbers in columns 3 and 4 of Fig. 8.

5.2 Detailed Results for *cook*

Due to lack of space, we are not able to present the results for all programs in this paper. Our technical report has the full details [33]. In this paper, we will present the detailed results for one program, namely, *cook*, and a summary of the results for the other programs. We chose *cook* because it is the second-largest program that all participants could analyze. *SNNS* is even larger and still was analyzed by all participants. However, results for *SNNS* were already presented during the First International Workshop on Detection of Software Clones (colocated with the 2002 International Conference on Software Maintenance) and the slides are available from [34] as well. Therefore, we focus on *cook* in this paper. The results for *cook* are representative for the other programs as well. Differences will be pointed out explicitly.

Unless stated otherwise in the following discussion, all values are given with respect to the good-metric and after 2 percent of the candidates were oracled.

Only two participants took the opportunity to send in a voluntary submission as well: Kamiya and Merlo. For both participants, unless stated otherwise, statements apply to both the mandatory and the voluntary data, which were very similar. We explicitly state differences between the mandatory and voluntary data if there are any.

5.2.1 Quality of the Detection

Fig. 11 shows the number of reported candidates by the participants. In one single aspect the program *cook* is exceptional: For the other seven programs, Kamiya reports more clones than all other participants. In the case of *cook*, however, Kamiya reports relatively few clones. The second most clones are usually reported by either Baker or Krinke. Looking at the clone type information, it can be seen that

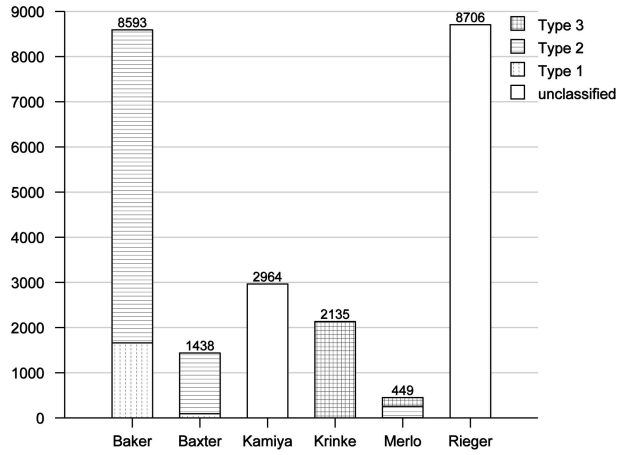

Fig. 11. Number of submitted candidates for program *cook*.

Krinke reports all his clones as type 3, whereas Kamiya and Rieger do not classify their clones at all. Baker and Baxter can identify type-1 and type-2 clones. Although it is hard to see in Fig. 11—because Merlo reported only very few type-1 clones in *cook*—he is the only one who is able to classify all three clone types.

In program *cook*, there are 402 reference pairs. In Fig. 12, you can see how the candidates match those references according to the ok-metric and the good-metric. It is interesting to note that Merlo, with only a fifth of the number of candidates that Krinke reported, manages to match more references than Krinke does. Another interesting point to note is the decrease of the matched references when looking at the ok-metric and the good-metric. Merlo loses almost no found reference when applying the stronger good-metric instead of the weaker ok-metric. This means that the clone candidates Merlo proposes are much closer to the real references than those of the other participants.

The distribution of how many references are jointly found by how many tools can be seen in Fig. 13. You can observe that a large number of the references is found only by one tool. And, in program *cook*, there was not even one single reference that was found by all tools.

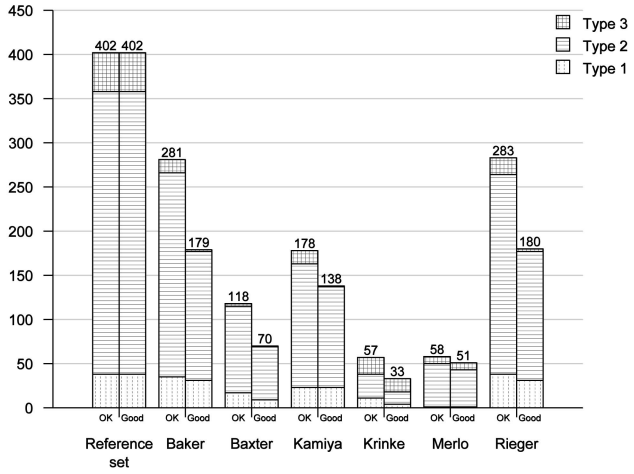

Fig. 12. Number of candidates matching references for program *cook*.

N	1	2	3	4	5	6
|Refs|	150	105	70	14	5	0

Fig. 13. Number of references of program *cook* that were matched by N tools.

In the source code of the program *cook*, we injected three secret clones:

1. One 40-line clone that is an identical copy of a sequence of statements in the same file.
2. A 28-line copy of a complete function across files where the parameter and all references to it in the function were renamed, thus resulting in a type-2 clone.
3. Another 22-line code fragment consisting of one complete function that was copied in the same file and two single-line statements were added, thus resulting in a type-3 clone.

The secret type-1 clone was found by Kamiya, Baker, and Rieger. All three found it according to the good-metric. The secret type-2 clone was found by all but Rieger and was found according to the good-metric. The type-3 clone was found by Kamiya, Baker, and Rieger, but only according to the ok-metric, not according to the good-metric, therefore, it is not accounted for in Fig. 14, which summarizes the number of secret references found by each tool.

The percentage of rejected candidates and recall are shown in Fig. 15. In general, tools that report a large number of candidates have a higher recall and a higher number of rejected candidates. For tools that report fewer candidates, the opposite is true. Only Krinke's tool has neither high recall nor high precision.

It is interesting to note that Baker's type-1 candidates are worse than her type-2 candidates, whereas the opposite is true for Baxter. The above-mentioned observation that Merlo's candidates fit the references very well is confirmed here again: Very few of his oracled candidates do not match a reference.

As we oracled only 2 percent of the submitted candidates, our values of recall and precision must be interpreted with caution. They can be compared among the different participants and the different programs, but are only relative numbers unless 100 percent are oracled.

5.2.2 Clone Size Aspects

In order to answer the question of how big clones tend to be (for our experiment, they had to be at least six lines long), we measured clone sizes as well. Fig. 16 shows the sizes of the code fragments for the individual tools. It is noteworthy that all three token-based techniques report clones that are longer than the clones of the other tools. Baxter and Merlo report by

Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
2	1	2	1	1	1

Fig. 14. Number of secret references found as good matches in program *cook* (three in total).

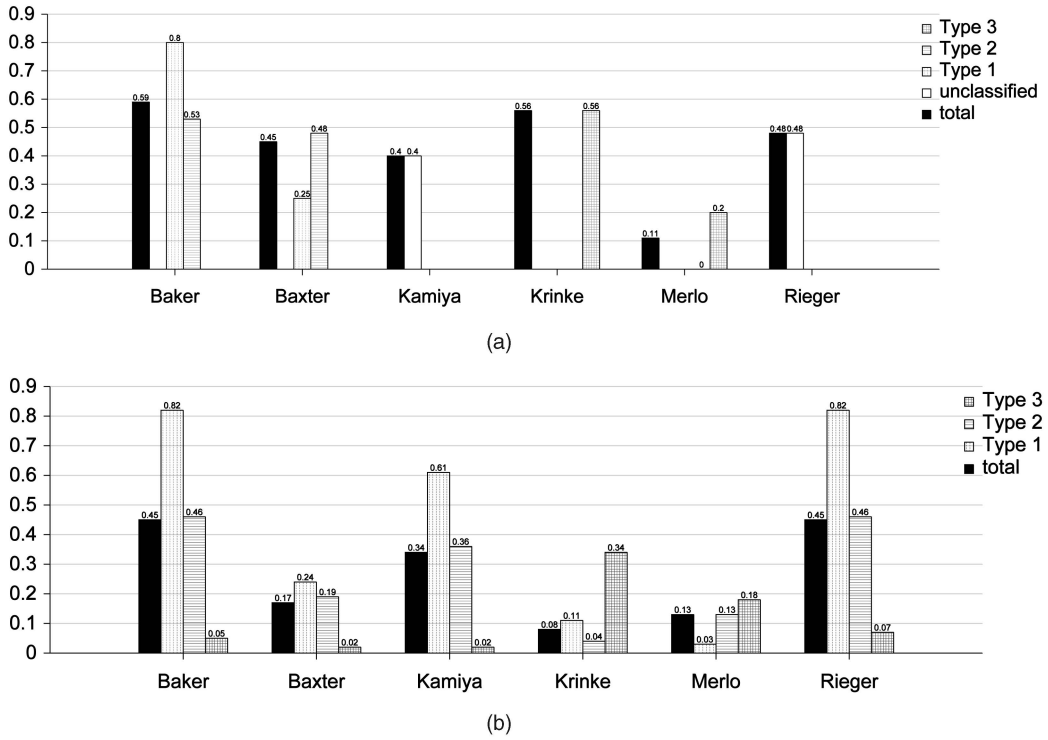

Fig. 15. Percentage of rejected candidates and recall for *cook*. (a) RejectedCands; see Definition 9. (b) Recall; see Definition 8.

far the smallest clones. This can be explained by the fact that Merlo explicitly looks for function clones and Baxter's clones must be able to get refactored by macro or function replacement. This again favors clones that are completely contained in one function. The token-based tools often detect code snippets that span more than one function.

5.2.3 Cloning Scope

Another interesting issue is whether more clones occur by cloning within the same file or by cloning across file boundaries (see Fig. 17). However, this is a program-specific value. Other programs have other ratios between intrafile clones and across-file clones. From the data we have, we can neither conclude a trend to intrafile nor to across-file clones as a general rule.

5.2.4 Distinctiveness of Tools

We wanted to know whether there are tools that find clones that no other tool can detect or whether there are tools that do not find particular clones that are otherwise found by all other tools. Those values are shown in Fig. 18.

In order to be able to refactor clones by replacing them with macros or functions, it is necessary that the code fragments of a clone pair do not overlap. The last column of Fig. 18 shows that not all tools pay attention to nonoverlapping. As a consequence, the tools by Baker, Kamiya, and Rieger cannot be used for automatic refactoring straightforwardly (regardless of precision of the candidates).

Participant	MaxCandSize or MaxRefSize	AvgCandSize or AvgRefSize	StdDevCandSize or StdDevRefSize
Baker	352	30.46	77.84
Baxter	28	9.37	4.24
Kamiya	207	18.85	23.95
Krinke	99	17.75	25.23
Merlo	26	10.04	3.39
Rieger	109	10.43	8.88
Reference set	645	22.41	60.70

Fig. 16. Sizes of code fragments in program *cook*.

Participant	|IntraFileCands| or |IntraFileRefs|	in %	|AcrossFileCands| or |AcrossFileRefs|	in %
Baker	40	22.35	139	77.65
Baxter	21	30.00	49	70.00
Kamiya	32	23.19	106	76.81
Krinke	2	6.06	31	93.94
Merlo	8	15.69	43	84.31
Rieger	36	20.00	144	80.00
Reference set	80	19.90	322	80.10

Fig. 17. Found references in the same file and in different files of program *cook*.

Participant	|OnlyRefs|	|OnlyButOneRefs|	|OverlappingCands|
Baker	32	0	243
Baxter	16	0	0
Kamiya	0	0	262
Krinke	21	2	0
Merlo	0	0	0
Rieger	46	1	165

Fig. 18. Further interesting values for program *cook*.

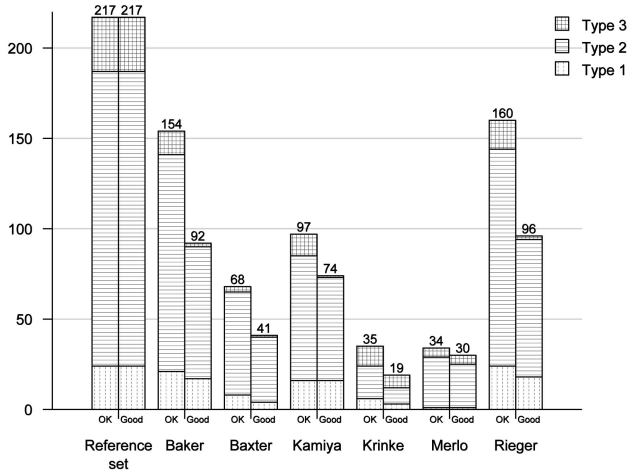

Fig. 19. Number of matching references for program *cook* after 1 percent of oracling.

5.3 Further Results

Having presented the detailed data for program *cook*, we will present selected data for different evaluations and other programs to underline certain observations or to show differences.

Fig. 19 shows the number of candidates matching references after only 1 percent of the candidates were oracled. It is noteworthy that if we compare this figure to Fig. 12, we do not see much difference with respect to the distribution of the bars. The only difference is the absolute number of candidates, which is almost a factor of 2 between Fig. 19 and Fig. 12.

So, despite having oracled only 2 percent of the candidates, it looks like, even with this small portion, the results are stable. This can be seen even better with the values of program *eclipse-ant* in Fig. 20: Even with such a small reference corpus, the relative results remain exactly the same.

Another interesting observation can be made when comparing the mandatory and the voluntary submissions of Kamiya and Merlo. There is almost no difference between the mandatory and voluntary submission for Merlo. Merlo maintains a high precision and a low recall for both submissions. In contrast, Kamiya managed to reduce his false positives with his voluntary submission,

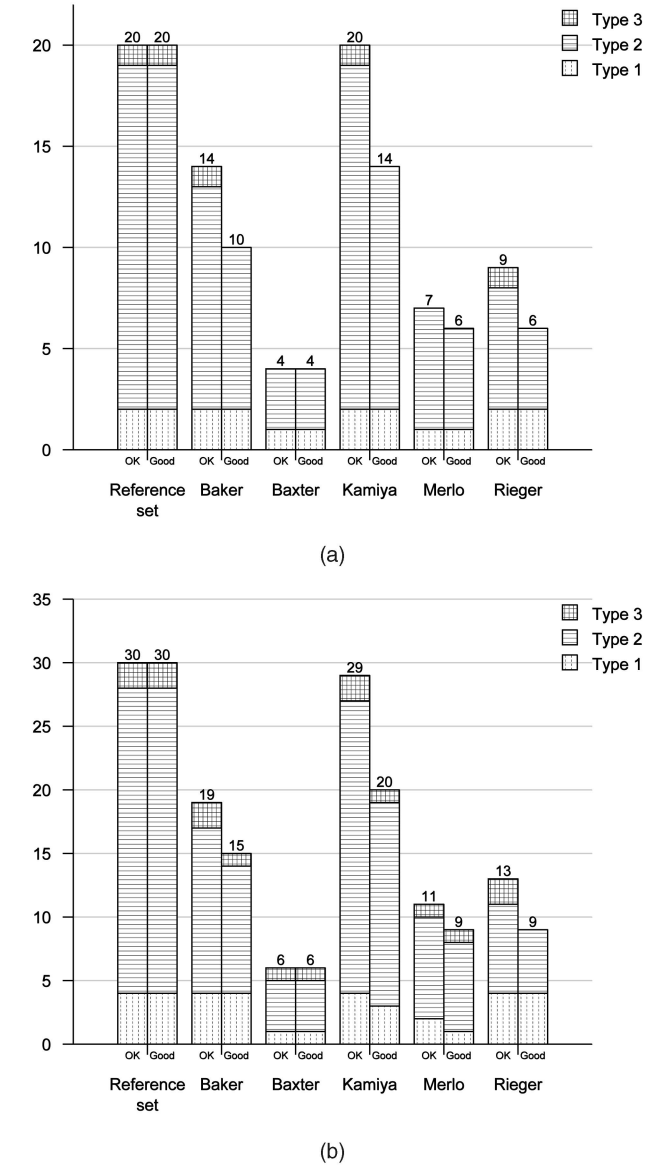

Fig. 20. Number of candidates matching references for program *eclipse-ant* (a) after 1 percent oracling and (b) after 2 percent oracling.

which is very obvious in the program *netbeans-javadoc*. Figs. 21 and 22 illustrate this reduction: He submitted only a forth of the candidates and still gets even better results.

In order to test the sensitivity of the results to the threshold p, we tried values for 0.6, 0.7, and 0.8. The absolute values of the results change as expected. The number of rejected candidates and true negatives increases with increasing p while recall and the number of found injected clones decrease. However, the relation among the results for the individual tools remains the same.

5.4 Clones Injected in All Systems

The fact that only between 24 percent and 46 percent of the injected secrets within all systems were found by the individual tools (ignoring Krinke, who found only 4 percent because he analyzed only three of the eight programs) needs more discussion. Seven of the injected clone pairs were not found by any of the participants. Of those clone

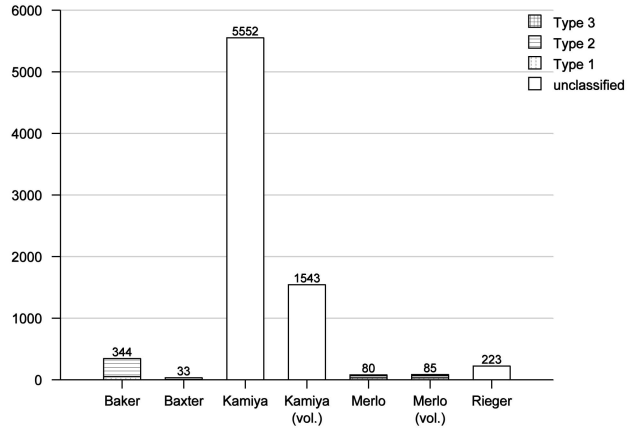

Fig. 21. Number of submitted candidates for program *netbeans-javadoc*.

pairs, one is a type-2 clone, the others are type-3. All injected type 1 clones were found.

There is no obvious explanation why the type-2 clone was missed. The missed type-3 clones were one method, one class declaration, one statement sequence, one try-catch block, and two struct declarations.

The cloned method had one declaration with initialization added, three statements added (`x++, --x, x = x > 15 ? 15 : x;`) at three different places, and variables renamed. Given the fact that the code fragments consisted of 10 or 14 lines, respectively, one can argue that the difference is substantial and, hence, it could be considered reasonable that the tools did not find that clone pair.

The class declaration had comments changed, one declaration of a local variable added, and a null expression replaced by a function call on the right-hand side of an assignment. These were also quite substantial changes in relation to the 11 lines.

The statement sequence of 12 or 13 lines, respectively, had one declaration with initialization added, a variable replaced by a literal, and two assignments added in the middle and at the end. These sequences was relatively similar.

In the try-catch block (in total, 9 and 11 lines, respectively), a method call was replaced by a string literal, an assignment was added, a simple assignment was turned into a declaration with initialization, a throw statement was added, and a package qualifier was extended. These were, again, relatively many changes.

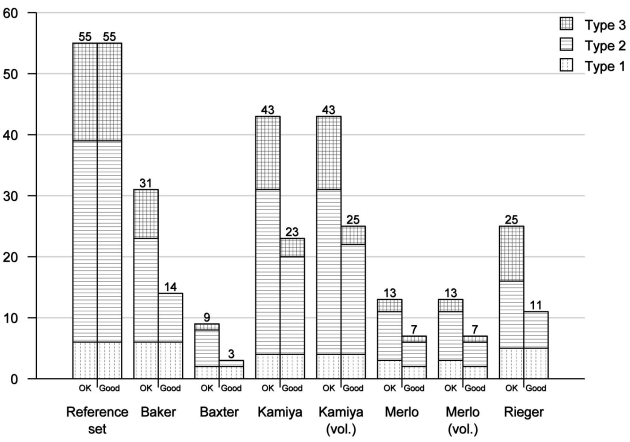

Fig. 22. Number of matching references for program *netbeans-javadoc*.

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
Speed	+ +	-	+	- -	+ +	–
RAM	+	-	+	+	+ +	–

Fig. 23. Performance comparison.

Two struct declarations overlapped in five identical components, and one struct had removed only one component. This change could have been found.

The other pair of struct declarations had two identical component declarations, three structurally identical, yet renamed components, one component added as second element, and two components added at the end. This pair was more different than the one described above.

In many of the missed cases, one can argue that the clones were too different. This is an inherent difficulty of the definition of type-3 clones. Yet, some were relatively similar from our point of view. One case shows a weakness of current clone detectors, which fail to unify different token sequences (even the AST-based tool) into the same syntactic category. While a string literal is rather different from a function call that returns a string, both are expressions of type string. The tools do not make this abstraction.

The fact that a substantial part of the injected secret clones were not found by the individual tools suggests that there are many other clones undiscovered in the programs.

5.5 Performance Aspects

Fig. 23 summarizes performance measures for the various tools. We note that the tools ran on different (yet ordinary) hardware platforms so that absolute numbers are not exactly comparable. We will mention the worst cases for each tool in the following.

The token-based techniques are very efficient. Baker's tool required less than 12 seconds and 62 MB and Kamiya needed 40 seconds and 47 MB for postgresql. Interestingly, Kamiya's tool is less efficient for Java; for j2sdk1.4.0-javax-swing it used 184 seconds and 44 MB (9 seconds, 50 MB for Baker). Yet, the most efficient tool was Merlo's. His tool never needed more than 4 seconds. Baxter's tool needed 3 hours and 628 MB for SNNS. Rieger's tool failed to analyze the largest C and Java systems. For SNNS, Rieger needed 840 seconds and 380 MB [35]. Similarly, Krinke's tool failed on postgresql (it analyzes only C). For Cook, it needed about 245 hours and 12 MB, and for SNNS about 63 hours and 64 MB.

6 Related Research

This section relates our evaluation to other similar studies.

Bailey and Burd compared three clone and two plagiarism detectors [36]. Among the clone detectors were three of the techniques we evaluated, namely, the techniques by

criterion	most suitable technique
suitability	metric-based
relevance	no difference
confidence	text-based
focus	no difference

Fig. 24. Assessment by Rysselberghe and Denmeyer.

Kamiya [9], Baxter [8], and Merlo [12]. Bailey and Burd used their own reimplementation of Merlo's technique; the other tools were original. The plagiarism detectors were JPlag [37] and Moss [38].

All clone candidates of the techniques were validated by Bailey and the accepted clone pairs formed a reference set against which the clone candidates were compared. Several metrics were proposed to measure various aspects of the found clones, such as scope (i.e., within the same file or across file boundaries), and the findings in terms of recall and precision were reported.

Our evaluation confirmed the study by Bailey and Burd. The syntax-based technique by Baxter had the highest precision (100 percent) and the lowest recall (9 percent) in this experiment. Kamiya's technique had the highest recall and a precision comparable to the other techniques (72 percent). Interestingly, Merlo's metric-based technique showed the least precision (63 percent), although in our study it is one of the most precise ones. The difference can be explained by the fact that Merlo compared not only metrics but also the tokens and their textual images to identify type-1 and type-2 clones in our study.

Although the case study by Bailey and Burd showed interesting initial results, it was conducted on only one relatively small system (16 KLOC). However, because the size was limited, Bailey was able to validate all clone candidates, while we were not because we dealt with many more and much larger systems. In both studies—Bailey's and ours—it is difficult to assess recall as none of us conducted an exhaustive manual search for clones. Our countermeasure was to inject clones that should be found.

The system Bailey analyzed was written by a postgraduate in Java. The systems we analyzed were written by multiple authors in Java or C, which allowed us to compare cloning in two different languages (although we could not identify any significant difference).

Other evaluations. While Bailey and Burd and our study focus on quantitative evaluation of clone detectors, other authors evaluated clone detectors for their fitness for a particular maintenance task. Rysselberghe and Demeyer [39] compared text-based, token-based, and metric-based clone detectors for refactoring. They compare these techniques in terms of suitability (Can a candidate be manipulated by a refactoring tool?), relevance (Is there a priority regarding which of the matches should be refactored first?), confidence (Can one solely rely on the results of the code cloning tool, or is manual inspection necessary?), and focus (Does one have to concentrate on a single class or is it also possible to assess an entire project?). They assess these criteria qualitatively based on the clone candidates produced by the tools. Fig. 24 summarizes their conclusions.

We should note that Rysselberghe and Demeyer originally state that the focus for suffix-tree based techniques is limited to smaller systems. They experienced a long runtime for a rather small system (11 KLOC C++) because of a high space demand on a 64 MB RAM machine. It is proven that suffix trees are linear in size of the program [7], and Baker's [1] and our own studies [40] showed that suffix-tree based clone detection scales very well in practice. The problem Rysselberghe and Demeyer encountered must be caused by their implementation.

Bruntink et al. use clone detection to find cross-cutting concerns in C programs with homogeneous implementations [41]. In their case study, they used `CCFinder` (Kamiya's [9] tool that we evaluated), one of the Bauhaus[1] clone detectors, namely `ccdiml`, which is a variation of Baxter's technique [8], and a PDG-based detector `PDG-DUP` [11]. The cross-cutting concerns they looked for were error handling, tracing, precondition and postcondition checking, and memory error handling. The study showed that the clone classes obtained by Bauhaus' `ccdiml` can provide the best match with the range checking, null-pointer checking, and error handling concerns. Null-pointer checking and error handling can be found by `CCFinder` almost equally well. Tracing and memory error handling can best be found by `PDG-DUP`.

7 Conclusion

It is difficult to determine a clear "winner" of this competition because all tools have their strength and weaknesses and, hence, are suitable for different tasks and contexts. Nevertheless, the comparison shed light on some facts that were unknown before. Strengths as well as weaknesses of the tools were discovered.

There are several important points to note when looking at the results of the comparison:

- The two token-based techniques and the text-based technique (Baker, Kamiya, and Rieger) behave astonishingly similarly.
- The tools based on tokens and text have higher recall.
- Merlo's tool and Baxter's AST-based tool have higher precision.
- The PDG-based tool (Krinke) does not perform too well (sensible only for type-3 clones).
- There is a large number of rejected candidates (between 24 percent for Baxter and 77 percent for Krinke).
- Many injected secret clones were missed (only between 24 percent and 46 percent of the injected secrets were found by the individual tools, ignoring Krinke who found only 4 percent because he analyzed only three of the eight programs).

The AST-based detection has a very high precision but currently has considerably higher costs in terms of execution time. The opposite is true for token-based techniques. If ideas from the token-based techniques could be made to work on ASTs, we would be able to find syntactic clones

1. http://www.axivion.com.

with less effort. In fact, the Bauhaus project has developed a combined technique along these lines since then. The combined technique uses suffix-tree based recognition on serialized ASTs. The resulting AST node sequences are then cut into syntactic units based on the AST structure [40]. Other combinations of the techniques should be explored further, too.

The syntax-based technique could be improved if they took more advantage of their syntactic knowledge. When we validated clone candidates, we often noticed fragments which are indeed clones from a purely syntactic point of view but are rarely meaningful, such as sequences of assignments, very long initializer lists for arrays, and structurally equivalent code with totally different identifiers of record components. Such spurious clones could be filtered out by syntactic property checks. Such a post-processing is described by Kapser and Godfrey [42] using lightweight parsing.

Type-1 and type-2 clones can be found reliably with existing techniques. Type-3 clones are more difficult to detect because they are inherently more vague. Here, in particular, we need to further explore meaningful definitions and types of similarity. Then, tools could be adjusted to find them more reliably. In particular, the tools currently lack abstraction. For instance, if two fragments are identical except that one uses a literal in several places where the other uses a function call, then current tools may be able to find the many type-1 and type-2 correspondences between the two fragments. Yet, they fail to notice that one fragment can be transformed into the other by replacing a literal by a function call consistently. Hence, they fail to identify the fact that the fragments are clones as a whole.

We used a human oracle to judge the clone candidates submitted by the tools. To avoid bias, an independent person (Stefan Bellon) investigated the candidates without knowing who submitted them (we were not developing clone detectors ourselves at the time of the experiment).

Still, the results are dependent on the judgment of Stefan Bellon. Because there is no definite definition of a clone, other judges might decide differently. Walenstein et al., for instance, report on differences among different human raters for clone candidates [43]. Yet, Walenstein et al. assumed a scenario in which clones ought to be removed. Their guidelines in the experiments suggested that clones should be detected worth removing. So, the sources of interrater difference could be the dissimilarity among clones or the appraisal of the need for removal. To see how much the results depend upon Bellon, we plan to replicate the experiment with different independent judges. Using independent judges may also help us to eventually reach a better consensus on what we consider clones in the first place. In particular, for only similar, not identical fragments, opinions differ.

Moreover, we will revisit our definition of a code fragment. In its current form, clones must consist of contiguous pieces of code, which puts Krinke's tool at a disadvantage. His tool takes only control and data flow into account and is independent of the textual order of statements, so it may report clones that need not consist of consecutive lines of code.

Another point of improvement that relates to the benchmark is to use token counts instead of lines as a measure of clone size. We often found clones that contained two statements separated by several blank or commented lines. In addition, generated files (like parsers) should be excluded from the benchmark because generated code tends to be regular and appears as spurious clone candidates.

The current benchmark requires a yes/no decision. It would be beneficial if human judges could express their confidence on a more refined ordinal scale. Finally, the benchmark should organize clones as equivalence classes for types 1 and 2 rather than clone pairs, which would ease the validation.

Because of limited space, only an excerpt of the results could be presented here (see [33], [44] for all results). The whole benchmark suite with source code of the comparison framework, the data submitted by the participants, the reference set, and evaluation results are available online at [34] so that the experiment can be inspected in detail, replicated, and enhanced for new systems and clone detectors. We hope the benchmark evaluation becomes a standard procedure for every new clone detector.

ACKNOWLEDGMENTS

The authors would like to thank all the participants in the experiment. They also would like to thank Magiel Bruntink and Brenda Baker for their comments on this paper. The authors also thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second Working Conf. Reverse Eng.*, L. Wills, P. Newcomb, and E. Chikofsky, eds., pp. 86-95, July 1995.

[2] K. Kontogiannis, R.D. Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," *Automated Software Eng.*, vol. 3, nos. 1-2, pp. 79-108, June 1996.

[3] B. Laguë, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," *Proc. Int'l Conf. Software Maintenance,* pp. 314-321, 1997.

[4] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance (ICSM '99),* 1999.

[5] J.H. Johnson, "Visualizing Textual Redundancy in Legacy Source," *Proc. Int'l Conf. Computer Science and Software Eng. (CASCON '94),* p. 32, 1994.

[6] B.S. Baker, "A Program for Identifying Duplicated Code," *Proc. 24th Symp. Interface,* pp. 49-57, Mar. 1992.

[7] B.S. Baker, "Parameterized Pattern Matching: Algorithms and Applications," *J. Computer System Science,* vol. 52, no. 1, pp. 28-42, Feb. 1996.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* 1998.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654-670, July 2002.

[10] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth Working Conf. Reverse Eng. (WCRE' 01),* 2001.

[11] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. Int'l Symp. Static Analysis,* pp. 40-56, July 2001.

[12] J. Mayrand, C. Leblanc, and E.M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance,* pp. 244-254, Nov. 1996.

[13] J.H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. Int'l Conf. Computer Science and Software Eng. (CASCON '93),* pp. 171-183, 1993.

[14] R.M. Karp and M.O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM J. Research and Development*, vol. 31, no. 2, pp. 249-260, Mar. 1987.
[15] E. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM*, vol. 32, no. 2, pp. 262-272, 1976.
[16] J.R. Cordy, T.R. Dean, and N. Synytskyy, "Practical Language-Independent Detection of Near-Miss Clones," *Proc. Int'l Conf. Computer Science and Software Eng. (CASCON '04)*, pp. 1-12, 2004.
[17] D. Gitchell and N. Tran, "Sim: A Utility for Detecting Similarity in Computer Programs," *Proc. 30th SIGCSE Technical Symp. Computer Science Education*, pp. 266-270, 1999.
[18] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On Software Maintenance Process Improvement Based on Code Clone Analysis," *Proc. Int'l Conf. Product Focused Software Process Improvement*, pp. 185-197, 2002.
[19] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo, "Pattern Matching for Design Concept Localization," *Proc. Second Working Conf. Reverse Eng., (WCRE '95)*, pp. 96-103, July 1995.
[20] G. DiLucca, M. DiPenta, and A. Fasolino, "An Approach to Identify Duplicated Web Pages," *Proc. Int'l Computer Software and Applications Conf. (COMPSAC '02)*, pp. 481-486, 2002.
[21] F. Lanubile and T. Mallardo, "Finding Function Clones in Web Applications," *Proc. Conf. Software Maintenance and Reeng.*, pp. 379-386, 2003.
[22] W. Yang, "Identifying Syntactic Differences Between Two Programs," *Software—Practice and Experience*, vol. 21, no. 7, pp. 739-755, July 1991.
[23] A. Marcus and J. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proc. Int'l Conf. Automated Software Eng.*, pp. 107-114, 2001.
[24] A.M. Leitao, "Detection of Redundant Code Using R2D2," *Proc. Workshop Source Code Analysis and Manipulation*, pp. 183-192, 2003.
[25] V. Wahler, D. Seipel, J.W. von Gudenberg, and G. Fischer, "Clone Detection in Source Code by Frequent Itemset Techniques," *Proc. Workshop Source Code Analysis and Manipulation*, pp. 128-135, 2004.
[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code," *Operating System Design and Implementation*, pp. 289-302, 2004.
[27] "Cook," http://miller.emu.id.au/pmiller/software/cook/, 2007.
[28] "The Stuttgart Neuronal Network Simulator," http://www-ra.informatik.uni-tuebingen.de, 2007.
[29] "PostgreSQL," http://www.postgresql.org, 2007.
[30] "Javadoc, http://javadoc.netbeans.org, 2007.
[31] "Eclipse," http://www.eclipse.org, 2007.
[32] "Java 2 SDK," http://java.sun.com, 2007.
[33] S. Bellon, "Vergleich von Techniken zur Erkennung duplizierten Quellcodes," master's thesis no. 1998, Universität Stuttgart, Germany, 2002.
[34] S. Bellon, "Detection of Software Clones—Tool Comparison Experiment," http://www.bauhaus-stuttgart.de/clones, 2007.
[35] S. Ducasse, O. Nierstrasz, and S. Demeyer, "On the Effectiveness of Clone Detection by String Matching," *J. Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37-58, Jan. 2006.
[36] J. Bailey and E. Burd, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *Proc. Second IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM '02)*, pp. 36-43, Oct. 2002.
[37] L. Prechelt, G. Malpohl, and M. Philippsen, "JPlag: Finding Plagiarisms among a Set of Programs," technical report, Univ. of Karlsruhe, Dept. of Informatics, 2000.
[38] S. Schleimer, D.S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proc. SIGMOD Int'l Conf. Management of Data*, pp. 76-85, 2003.
[39] F. Van Rysselberghe and S. Demeyer, "Evaluating Clone Detection Techniques from a Refactoring Perspective," *Proc. Int'l Conf. Automated Software Eng.*, 2004.
[40] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *Proc. Working Conf. Reverse Eng.*, 2006.
[41] M. Bruntink, R. van Engelen, and T. Tourwe, "On the Use of Clone Detection for Identifying Crosscutting Concern Code," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 804-818, Oct. 2005.
[42] C. Kapser and M. Godfrey, "Improved Tool Support for the Investigation of Duplication in Software," *Proc. Int'l Conf. Software Maintenance (ICSM '05)*, pp. 305-314, 2005.
[43] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems Creating Task-Relevant Clone Detection Reference Data," *Proc. Working Conf. Reverse Eng.*, 2003.
[44] S. Bellon, "Vergleich von Techniken zur Erkennung Duplizierten Quellcodes," master's thesis, Univ. of Stuttgart, Germany, Sept. 2002.

Stefan Bellon received the degree in computer science from the University of Stuttgart in 2002, where he was a student at the time of the experiment reported in this paper. He is managing director and one of the founders of Axivion GmbH, a spin-off of the Bauhaus research project, offering tools and services to better support software maintenance and evolution. His research interests are in clone detection, program analysis, and software maintenance.

Rainer Koschke received the doctoral degree in computer science from the University of Stuttgart in Germany. He is a professor of software engineering at the University of Bremen in Germany. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, clone detection, and reverse engineering. He is a member of the IEEE Computer Society.

Giuliano Antoniol received the degree in electronic engineering from the Universitá di Padova in 1982. In 2004, he received the PhD degree in electrical engineering from the Ecole Polytechnique de Montreal. He has worked for companies, research institutions, and universities. In 2005, he was awarded the Canada Research Chair Tier I in Software Change and Evolution. He is currently a full professor at the Ecole Polytechnique de Montreal, where he works in the areas of software evolution, software traceability, and search-based software engineering. He is a member of the IEEE.

Jens Krinke received the PhD degree in computer science from the University of Passau, Germany, in 2003. He is now an assistant professor of software technology at Fern Universität/University in Hagen. He has worked on various aspects of program slicing, in particular on context-sensitive program slicing of concurrent programs. Other research interests include clone detection, aspect mining, and distance teaching of software engineering through computer-supported collaborative learning. He is a member of the IEEE Computer Society.

Ettore Merlo received the PhD degree in computer science from McGill University (Montreal) in 1989 and the laurea degree from University of Turin (Italy) in 1983. He led the software engineering group at the Computer Research Institute of Montreal (CRIM) until 1993, when he joined Ecole Polytechnique de Montreal as a professor. His research interests are in software analysis, software reengineering, user interfaces, software maintenance, artificial intelligence, and bioinformatics. He is a member of the IEEE.