Finding Clones with Dup: Analysis of an Experiment

Brenda S. Baker, *Member*, *IEEE*

Abstract—An experiment was carried out by a group of scientists to compare different tools and techniques for detecting duplicated or near-duplicated source code. The overall comparative results are presented elsewhere. This paper takes a closer look at the results for one tool, Dup, which finds code sections that are textually the same or the same except for systematic substitution of parameters such as identifiers and constants. Various factors that influenced the results are identified and their impact on the results is assessed via rerunning Dup with changed options and modifications. These improve the performance of Dup with regard to the experiment and could be incorporated into a postprocessor to be used with other tools.

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

1 Introduction

Software systems often have sections of code that are similar, as a result of informal code reuse (copying and pasting). Once created, the initially similar regions may be maintained separately and undergo independent modifications. It may be useful to identify such sections to make sure that bugs fixed in one copy get fixed in another, to refactor the code to eliminate redundancy, or to identify plagiarism.

Dup [1], [2], [3], [4] is a tool that finds maximal pairs of sections of code that are textually similar, where similar may mean either exactly the same or exactly the same except for systematic substitutions of parameters such as identifiers and constants, e.g., the substitution of `y` and `fun` for `x` and `foo`, respectively, everywhere they occur. The latter type of similarity is termed a *parameterized match* or *p-match*. Based on an efficient data structure, Dup is designed to analyze large systems, even with millions of lines of code.

Dup was part of an experiment that compared several tools for finding duplicated or near-duplicated regions, termed clones, in C and Java source code. Rainer Koschke proposed the experiment and invited a number of tool-owners to participate. Stefan Bellon designed the experimental method in consultation with the tool-owners and built software to evaluate the results. The participants included the author of this paper with Dup, Ira Baxter with CloneDR [5], Toshihiro Kamiya with CCFinder [6], Jens Krinke with Duplix [7], Ettore Merlo with CLAN [8], [9], and Matthias Rieger with Duploc [10]. The tools are based on various techniques, including token-based analysis (Dup and CCFinder), line-based analysis (Duploc), abstract syntax trees (CloneDR), program dependency graphs (Duplix), and metrics (CLAN).

• *The author can be reached at 140 North Road, Berkeley Heights, NJ 07922. E-mail: brendabaker@ieee.org.*

Manuscript received 15 Apr. 2006; revised 23 Oct. 2006; accepted 31 May 2007; published online 27 June 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-0095-0406.
Digital Object Identifier no. 10.1109/TSE.2007.70720.

The experiment challenged each tool to find clones of three types:

1. Type 1: exact,
2. Type 2: the same except for changed parameters such as identifiers, constants, or types, and
3. Type 3: copy with modifications (additions, deletions, and other changes).

Bellon et al. [11] describe the problems that had to be overcome in order to find a common ground for an experiment with these disparate tools, the experimental method eventually chosen, and data comparing the various tools.

Briefly, the experimental method was as follows (definitions and details are given in Section 3): The tools were used to analyze four systems written in C and four systems written in Java to find the three types of clones, subject to a minimum size of six lines. The clones reported by the tools are termed *candidates*. Bellon randomly selected 2 percent of the candidates and *oracled* each one (called a *seen-candidate*) by looking at it, deciding whether he thought the two regions specified by the candidate represented a clone, and, if so, determining the boundaries that he thought were appropriate for the resulting clone, called a *reference*. A methodology for comparing candidates and references and a comparison of the tools with regard to one system are presented in [11].

Since the tools varied in terms of what types of candidates they could generate (and even in whether they could process all the systems) and Bellon [12] provided summaries of the data for many possible measures of performance, a comparison of the tools is not straightforward. However, Dup came out well in the experiment. In particular, under the measure of recall (percent of references "found") defined in this paper, Dup came in first or second out of all the tools on seven of the eight systems and third on the eighth, both for recall of Type 1 references and recall of Type 2 references. In terms of the proportion of Type 1 and 2 candidates Bellon rejected (as defined in this paper) in oracling, Dup ranged from first to sixth best among the tools on the eight systems, with an average of 3.4th. In general, for Type 1 and 2 references, the three

token-based or line-based tools (Dup, CCFinder, and Duploc) tended to have higher recall, while CLAN (based on metrics) and CloneDR (based on abstract syntax trees) generated relatively fewer candidates compared to the number of references "found." Duplix (based on program dependency graphs) reported only Type 3 clones [11].

It is not easy to derive an understanding from the raw data for why a tool did well or poorly with regard to the measures that Bellon established. Since a major impact on the results came from the particular oracling decisions made by Bellon, the first problem is to understand how the oracling was done. While Bellon did not formalize an oracling algorithm, discussions with Bellon and examination of seen-candidates and references revealed that his oracling was largely systematic. This paper gives a set of guidelines that seem in large part to describe his oracling. The paper also describes some situations in which the oracling appeared inconsistent, mainly in cases where the possible references would most likely be of marginal interest for a user.

The next problem is to understand the reasons that a tool did or did not do well for various aspects of the experiment. Since there were huge numbers of candidates and references, it would be impossible to look at all of them by eye individually. The only hope is to identify some relevant factors and then to automate an analysis of how they affected performance. This paper does this for Dup; a similar methodology could be used for other tools.

A major question is whether the results from the experiment are meaningful in comparing the capabilities of the various tools and the various techniques they are based on. In particular, to what extent did the results for individual tools depend on the core algorithms used versus the interaction of the design and execution of the experiment with other choices, such as how to treat various features of C and Java or how to define the boundaries of a clone, that might be modifiable for an individual tool, selectable by tool options, or fixable by a postprocessor? This paper shows that the results were significantly affected by such choices, at least for Dup. The same may be true for the other tools.

In particular, this paper analyzes the effect of various factors on Dup's performance with regard to recall (percentage of references discovered by Dup) and rejection (percentage of Dup candidates that were oracled and rejected by Bellon). (Definitions are given in Section 3.) Discussion is limited to Types 1 and 2, as Dup did not generate Type 3 candidates for the experiment.[1] Evaluation is via a combination of simple options added to Dup and other software created for the purpose.

Factors for which the effects on Dup's recall are individually estimated include a default filtering option in Dup, differing definitions of Type 2, oracling of clone boundaries based on nesting level of statements, and different choices by different tools for how to report clones in repetitive regions of code. (A repetitive region is of the form $A_1A_2...A_n$, where each A_i is a clone of A_j, for $i \neq j$.) Other factors include layout differences (including some introduced by Bellon's reformatting of code for the experiment), comments (counted toward clone size by Bellon but not Dup), substitution of expressions or types (allowed by Bellon) rather than single identifiers and constants (as required for Dup), and Bellon's taste in oracling. No one factor is dominant for either C or Java.

Factors whose effect on rejection are individually estimated include oracling of clone boundaries based on nesting level of statements, Dup's allowing preprocessor statements (in C) and import statements (in Java) to be part of a clone, and table initializations. Other factors affecting rejection include Bellon's taste in determining ending boundaries of clones versus Dup's reporting of maximal clones, and other oracling choices made by Bellon. Again, no one factor is dominant for either C or Java.

The results in this paper show that much of the effect of these factors could have been avoided or alleviated without changing the core algorithms of Dup. The same may be true of other tools. The reason is that the oracling was largely systematic and based on textual and syntactic features. For the analysis, many of the oracling decisions were incorporated into Dup via small modifications or choice of options. In fact, many of the oracling decisions could be incorporated into a postprocessor that could be applied to the candidates of any of the tools, with possible improved performance. With such a postprocessor, the overall results might change in terms of absolute and relative performance of the tools. Hence, it cannot be concluded that the absolute or relative differences found in the experiment are necessarily inherent to the various tools or to the core techniques they are based on.

On the bright side, the systematic nature of the oracling means that it could be incorporated into a useful tool. Such a tool might be used in future experiments to reduce the impact of the sorts of factors mentioned above. In fact, it suggests a possible new category of clone-related tool, a *clone-massager*, that would postprocess candidates. A particular clone-massager could be designed to suit a particular person's taste in defining clones, a particular application, or both.

This paper also studies how repetitive regions can impact the results. The impact can be large due to the large number of candidates and references that may result from such regions and the different possible methods for reporting candidates in such regions.

The remainder of the paper is organized as follows: Section 2 describes Dup and how it finds clones. Section 3 describes the experimental setup (first as it was presented ahead of time to the participants and then as it was realized), the methodology for analysis of results, and the recall and rejection for Dup. In Section 4, an analysis is made of how recall was affected by various factors, and repetitive regions are discussed. The factors affecting rejection are analyzed in Section 5, and oracling is further discussed with regard to particular code characteristics that it allowed or disallowed. Section 6 presents a retrospective evaluation of the experimental design and the properties of the particular systems that were analyzed by the tools, including the effect of repetitive code. Finally, Section 7 discusses the overall contributions from the experiment, proposes modifications to the experimental method for future experiments, and suggests possibilities for future tools and areas for future research.

2 Description of Dup

Dup finds either maximal exact matches over a threshold length or maximal parameterized matches over a threshold

1. A Type 3 option was included in an earlier version of Dup [2] but was dropped from more recent versions.

length, where the threshold length is specified by the user. The threshold length is the smallest clone size that Dup will report, measured in terms of noncommentary source lines (*ncsl*) in each fragment. Exact matches, parameterized matches, and maximality are defined below.

Dup does not parse the input. However, Dup tokenizes the input and classifies each token as a parameter or nonparameter. Tokenization depends on the language being processed. Identifiers and constants are considered parameters, while keywords and punctuation are nonparameters. Comments, blank lines, and white space within lines are ignored. For efficiency in processing huge software systems, Dup is semi-line-based: its tokenizers produce one token per parameter, but only one nonparameter token for the remainder of each line. To obtain the nonparameter token, Dup constructs a string representing the nonparameter part of the line and applies a hash function. For example, in a line $zap = 20 * foo(zap);$, there would be four parameter tokens representing the first occurrence of zap, 20, *foo*, and the second occurrence of zap. The remainder of the line can be represented by a string $P = P * P(P);$ (with no white space), where each P is a placeholder for a parameter and one nonparameter token is obtained by hashing this string. The whole line can be represented by the single nonparameter token followed by the list of parameter tokens for the line. The output is line-based; partial lines are truncated.

A string of tokens (also called symbols) is termed a *parameterized string* or *p-string*. Two p-strings are an *exact match* if they contain exactly the same symbols. Two p-strings are a *parameterized match* (*p-match*) if there is a one-to-one function on the alphabet of parameter and nonparameter symbols such that the function maps each nonparameter symbol into itself and each parameter symbol into a (possibly different) parameter symbol, and one string is mapped by this function into the other. For example, if x and y are parameter symbols and a and b are nonparameter symbols, string $axax$ p-matches $ayay$ (with x mapped into y and a into itself) but not $axay$ or $byby$.

For dealing with p-matches, the following encoding for p-strings is useful: Each occurrence of a parameter symbol is replaced by a 0 if it is the first occurrence and by the distance (in number of tokens) from the previous occurrence otherwise. Nonparameter symbols are left unchanged. For a p-string S, this encoding is called $prev(S)$. For example, if x and y are parameter symbols while a and b are nonparameter symbols, then $xxaybxy$ would be encoded as $01a0b43$. The first 0 represents x, while the second 0 represents y; the positive integers encode the other occurrences of x and y. (In examples, all distances will be single digits so that this notation will be unambiguous.) Note that, if u and v are also parameter symbols, then $uuavbuv$, which p-matches $xxaybxy$, would have the same encoding. Thus, p-string S p-matches p-string T if and only if $prev(S) = prev(T)$.

If two substrings of a p-string S are an exact match and the exact match cannot be extended in either direction, i.e., one symbol to the left or one symbol to the right, then the match is a *maximal exact match*. Similarly, if a parameterized match cannot be extended to the left or right (while maintaining the same mapping of any parameter tokens within them), it is a *maximal p-match*. The data structures and algorithms for finding maximal exact matches and maximal p-matches are described briefly in this paper; details and proofs of correctness are given in [1], [3], [4].

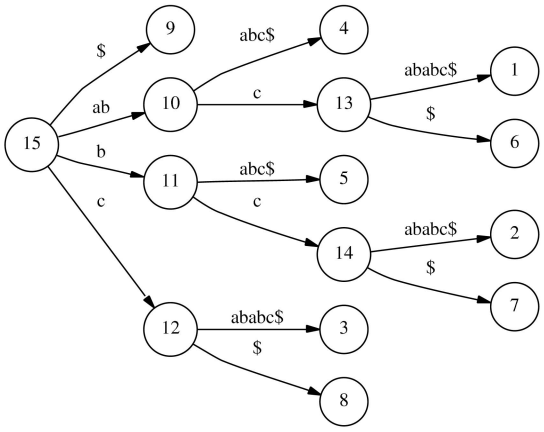

Fig. 1. A suffix tree for $abcababc\$$.

2.1 Finding Maximal Exact Matches

First, we describe how to find maximal exact matches in an input string via a suffix tree. A suffix tree [13], [14] is a compacted trie for the suffixes of the input, where the suffixes of an input string $a_1a_2...a_n\$$ are $a_i...a_n\$$ for $1 \leq i \leq n$. (For technical reasons, the input string must be terminated by an end marker, \$ in this example, that does not occur elsewhere in the string.) The trie is compacted in that there are no nodes with just one child; instead, an edge can be labeled with a string rather than just a character. The tree can be stored in linear space if the substring labels are represented by pointers into the input string.

An example of a suffix tree for a string $S = abcababc\$$ is shown in Fig. 1. Each leaf corresponds to a suffix in the sense that concatenating the symbols on the path from the root to that leaf yields the suffix; in the figure, the leaf labeled i corresponds to the suffix starting at position i, for $1 \leq i \leq 9$.

A suffix tree directly encodes part of the information needed to identify maximal matches. In a suffix tree, two suffixes share a path from the root to the point at which the suffixes diverge, i.e., the point at which the next symbols are different. For example, in our figure, suffixes 1 and 6 share a path from the root to node 13, with label abc, at which point the suffixes diverge; similarly, suffixes 2 and 7 share a path until node 14, with label bc.

However, the suffix tree does not directly encode whether the preceding symbols are different. For example, the structure of the suffix tree in the figure does not show that the two occurrences of abc are a maximal match, while the two occurrences of bc are both preceded by a and are not a maximal match. For a string $T = s_1s_2...s_n$, let $T[i,j] = s_i...s_j$. For a substring $T[i,j]$, where $1 \leq i,j < n$, the left context is s_{i-1} (or a dummy symbol if $i = 1$), and the right context is s_{j+1}. For a path from the root to an internal node, the suffix tree shows the right context but not the left context, and both are needed to identify maximal matches.

The maximal matches can, however, be found by computing the left context dynamically when needed. At each node n, the goal is to report a maximal match for each instance in which a suffix for a leaf from one child's subtree has left context s_1 while a suffix for a leaf from a different

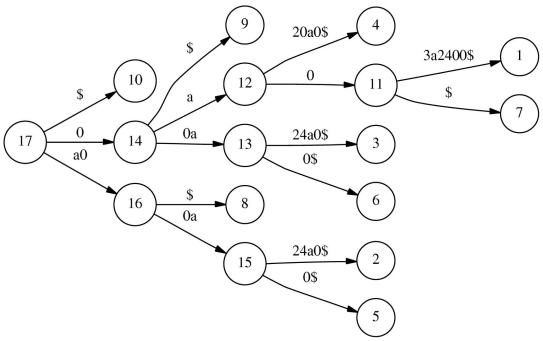

Fig. 2. A p-suffix tree for $xayxaxyaz\$$, where x, y, and z are parameters and a is not.

child's subtree has left context s_2, $s1 \neq s_2$; the starting positions of the matching substrings are the same as the starting positions of the suffixes corresponding to the two leaves, and the match length is the length of the concatenated sequence of labels from the root to n.

This can be done recursively. In the course of the recursion, the suffixes corresponding to descendant leaves of each node are classified into lists according to left context, with one list for each left context symbol. More precisely, a *p-list* is a list of starting positions of suffixes with the same left context; a *c-list* is a list of p-lists. For a node n, the algorithm is first called recursively on its children, so that the c-lists of the children are available. Next, for children c_1 and c_2, $c_1 \neq c_2$, and p-lists lst_1 for c_1 and lst_2 for c_2, if the left context of lst_1 is different from the left context of lst_2, the algorithm reports a maximal match for each pair of positions in the cross-product of lst_1 and lst_2. Finally, it accumulates its own p-lists by processing the children one by one and merging the p-lists with the same left context. If it is desired to report only those maximal matches over a threshold length, the algorithm only processes the nodes for which the concatenated sequence of labels from the root is sufficiently long.

For fixed alphabets, building the suffix tree takes space and time linear in input length [13]. Recursing over the suffix tree takes space linear in input length and time linear in the input length plus the number of matches [1].

2.2 Finding Maximal p-Matches

The above method can be generalized to find maximal p-matches by means of the $prev$ encoding and a new data structure called a *parameterized suffix tree* (*p-suffix tree*). A p-suffix tree for a p-string S is a compacted trie for $prev(S_i)$, $1 \leq i \leq |S|$, where S_i is the ith suffix of S. (As before, S is assumed to end in a character that does not occur elsewhere in the string.) For example, let $S = xayxaxyaz\$$, where x, y, and z are parameters and a is a nonparameter. Then, $S_1 = S$ and $prev(S_1) = 0a03a24a0\$$, $S_2 = ayxaxyaz\$$, $prev(S_2) = a00a24a0\$$, and so forth. Note that the 3 in $prev(S_1)$ changed to a 0 in $prev(S_2)$. A p-suffix tree for S is shown in Fig. 2. The label on the path from the root to leaf i is $prev(S_i)$, for $1 \leq i \leq |S|$.

Recursing over the p-suffix tree to report maximal p-matches can again be done recursively. However, the details are more complicated than before. For the left context of a substring of T, we cannot simply use the value of the preceding symbol in $prev(T)$. For example, if $T = ubvwbw$, where u, v, and w are parameters and b is a nonparameter, bv and bw match because $prev(bv) = b0 = prev(bw)$. The first u and the first w become 0 in $prev(T)$, suggesting that the p-match can be extended to the left, whereas it cannot, because $prev(ubv) = 0b0$ while $prev(wbw) = 0b2$. For a p-string $T = s_1 s_2 ... s_n$, and $1 \leq i, j \leq n$, let $T[i, j] = s_i s_{i+1} ... s_j$. Our solution is to use the reverse of the $prev$ function on whatever substring is being considered. In particular, for $T[i, j]$, the left context is s_{i-1} if s_{i-1} is a nonparameter, and otherwise, 0 if s_{i-1} does not occur in $T[i, j]$, and k if the next occurrence of s_{i-1} is at position $i - 1 + k$.

In the recursion over a p-suffix tree, obtaining the p-lists for a parent from those of the children is more complicated than before. For suffix trees, to obtain the list of leaves with left context symbol a at node n, the lists for a of the children can simply be merged. For p-suffix trees, in going from child to parent, the substring under consideration gets shorter, one or more parameter symbols may be truncated, and a nonzero left context may change to zero. The recursion has to merge the lists so as to take such changes into account. In Fig. 2, an example of such a change is at node 16, where the left context of 3 for the suffix S_2 at node 2 changes to zero at node 16, whereas the left context of 2 for the suffix S_5 at node 5 remains 3 at node 16. The left context of the suffix of node 8 is 0. Thus, at node 16, we discover that $S[2, 3] = ay$ and $S[8, 9] = az$ are not a maximal p-match because both have left context 0.

Constructing a p-suffix tree is also more complicated than constructing a suffix tree. McCreight's algorithm for constructing a suffix tree fails to construct a p-suffix tree because p-strings lack a fundamental property of strings needed for McCreight's algorithm to work. The algorithms for constructing p-suffix trees and finding maximal p-matches are given in [1], [3], and [4].

2.3 Dup Options and Performance

Dup allows the user to set the threshold size (in lines) for reporting maximal matches. The user can ask Dup to find maximal p-matches or only maximal exact matches; in the former case, Dup reports when a p-match is also an exact match. Dup has an option to filter out p-matches that have too many changed parameters. By default, Dup filters out p-matches that have more than half as many changed parameters as lines because such instances often seem less related than when there are few changed parameters. Since the number of p-matches found in large software systems is often enormous, the filtering option can be used to eliminate some of the less useful matches. Dup also has a default option to truncate any initial lines consisting of closing braces. Additional options were added to Dup for the analysis in this paper; they are described where used in the analysis.

Dup is implemented in about 4,000 lines of C and runs under UNIX. It is very fast. For example, on the largest project analyzed in this experiment, with 202 K source lines and 136 K noncommentary source lines, Dup used 11.7 seconds and 57 MB when run on one processor of an IRIX machine with eight 250 MHz processors.

3 The Experiment

This section describes the experiment as it was presented to the tool-owners beforehand and as it turned out in actuality. Then, it gives the definitions needed for evaluation of

results, briefly describes the evaluation method used by Bellon et al. [11], and, finally, presents the evaluation method used in this paper.

3.1 The Plan for the Experiment

In order to induce owners of independently designed, disparate tools to join the experiment, the initial description of the experiment was quite informal. The word "clone" was never defined. The three types of clones to be found were also not precisely defined, but merely described as "Type I: exact," "Type II: parameterized copy," and "Type III: copy with modifications (additions, deletions, ...)." It was not specified whether to report clones of one type that were also part of longer clones of another type. These choices were left to the tool owners. The goal was to define the experiment in such a way that it would not exclude any of the tool techniques.

However, the participants agreed to report clones in a line-based manner, although some of the tools were not line-based. Moreover, they agreed not to report clones shorter than six lines, based on the starting and ending line number of each clone. Finally, they agreed that comments and layout (such as tabs and line breaks) should be ignored, and that the tools should not look at C header files included via preprocessor lines.

Because of differences in how various tools reported clones, Bellon reformatted ("normalized") the input to try to negate the effects of some of the differences. He deleted lines consisting of white space (but not lines consisting only of comments). He reformatted the input to move braces appearing alone on lines to the end of the preceding line; if there were several such lines in succession, braces accumulated at the end of the previous line of code.

The participants agreed that Bellon would select a random subset of the candidates submitted and would act as an "oracle" to decide whether they were valid clones. He declined to specify ahead of time on what basis he would determine if a candidate was a valid clone.

After an initial test run on two systems written in C and two systems written in Java, Bellon asked the participants to run their tools on four systems written in C and four systems written in Java. The systems used as input for the experiment are termed *projects* and are listed at [12]. In total, the C projects contained 797 files, 377,021 lines of code, and 261,279 noncommentary source lines (ncsl), while the Java projects contained 1,558 files, 400,775 lines of code, and 226,634 ncsl.

3.2 Realization of the Experiment

This section describes the experiment as it was actually executed. The vagueness of the description of the plan for the experiment left room for surprises in how the oracling was done. The main surprise for the author of this paper was the definition Bellon intended for Type 2 clones.

The three types that Bellon listed sounded like the three types in [2], except that [2] used the term "match" rather than "clone." In particular, Bellon's description of a Type 2 clone as a "parameterized copy" sounded like the parameterized matches (p-matches) discussed in Section 2. This definition of parameterized match has also been adopted by others in the algorithms community [15], [16], [17].

In fact, Bellon did not have in mind requiring that each occurrence of a parameter of one fragment be renamed as the same parameter in the other fragment; his intention was that different occurrences of a parameter could be renamed differently, e.g., an `x` could be renamed as `y` in one place and `n1` in another. He also allowed an expression to be substituted for a single parameter. The other participants apparently understood this version of the definition.

Henceforth, Type 2 will refer to Bellon's definition, in contrast to p-matches. In [11], the definition was restated as *syntactically identical copy; only constants and/or variable, type, or function identifiers have been changed*.

The ambiguity affected the experiment in that Dup could have been used to generate Type 2 candidates (more precisely, Type 2 candidates limited to substitution of single parameters for single parameters) if the definition that Bellon intended had been clear. In fact, the original implementation of Dup, before the invention of p-suffix trees, found such Type 2 matches and postprocessed them to identify p-matches. Section 4.2 analyzes the impact of the differing definitions on the results for the experiment by means of putting a Type 2 option back into Dup.

Bellon oracled 2 percent of the candidates produced by the various tools to produce the set of reference candidates (*references*). He did not establish a formal set of principles for oracling. Nevertheless, his oracling was largely systematic. The following informal guidelines (based on inspection of candidates and references and e-mail discussion with Bellon) describe how Bellon extracted references from oracled candidates, called *seen-candidates*.

Oracling Guidelines:

1. A reference must have at least six lines in each fragment, including commentary lines in the middle but not the beginning or the end.
2. A reference cannot end at a lower level of nesting than that of the first line of the reference, e.g., it cannot start in the middle of a `while` loop and continue to statements after the `while` loop.
3. A Type 2 clone may include substitution of expressions for variables or constants, e.g., `&p->c` for `p`.
4. A reference should be a maximal clone of that type, i.e., it cannot be expanded forward or backward to a longer clone of the same type, subject to the other guidelines.
5. For repetitive regions, where reporting styles differ for different tools, the reference should be in the same style as the one the tool proposed.

For each seen-candidate, Bellon applied the guidelines to obtain a possible reference whose boundaries might differ from those of the seen-candidate. He might then generate a reference, or he might reject it based on other characteristics of the code fragments.

These guidelines formed the starting basis for the analysis in this paper. Other observations on oracling arose during analysis, and are discussed where relevant, especially in Section 5, which discusses language and structural features that seemed to influence whether Bellon generated references from candidates, although not necessarily consistently.

3.3 Evaluating the Results

The task of finding clones in a large set of code bears some similarity to finding a set of documents in response to a query in information retrieval. The analogy is not exact, but

information retrieval literature is helpful in how to analyze the data.

For information retrieval, results are generally stated in terms of recall and precision. The space of all documents can be classified in four groups:

- retrieved and relevant,
- retrieved and irrelevant,
- not retrieved and relevant, and
- not retrieved and irrelevant.

Recall is defined as the number of relevant documents retrieved divided by the number of all relevant documents, whereas precision is the number of relevant documents retrieved divided by all documents retrieved.

For this experiment, the only possible set of "relevant documents" is the set of references found by Bellon through oracling 2 percent of the candidates. "All documents retrieved" corresponds to the set of all candidates from a tool. Hence, recall should be the percentage of references discovered by a tool and precision should be the number of references found divided by the number of candidates.

However, for this paper, we need to stick to measures that can be reevaluated without further oracling, as the plan is to rerun Dup with modifications to see how the performance changes based on various factors. For a changed (and especially a larger) set of candidates, precision cannot be fairly evaluated without additional oracling of new candidates to generate more "relevant" documents. Therefore, our approach is somewhat different from usual. The two measures studied in this paper are *recall* and *rejection*, where rejection is the percentage of seen-candidates that were rejected as clones by Bellon.

3.4 Defining Recall and Rejection

Defining recall and rejection precisely depends on defining when a reference has been "found" by a tool or "rejected" in oracling. In a preliminary test run, there were few instances in which the same pair of ranges of lines were reported as candidates by two different tools. Moreover, Bellon often changed the boundaries in extracting a reference from a candidate. Hence, Bellon et al. [11] developed a formula for evaluating the goodness of a match between a reference and a candidate based on the amount of overlap of the two pairs of fragments.

Following Bellon et al. [11], we define a code fragment as a triple (f, s, e), where f is the filename, s is the starting line number, and e is the ending line number. If CF is a code fragment, the parts of the triple can be designated as $CF.filename$, $CF.StartLine$, and $CF.EndLine$.

The *overlap* of two code fragments f_1 and f_2 is defined to be $overlap(f_1, f_2) = |f_1 \cap f_2|/|f_1 \cup f_2|$. For example, if the first code fragment is $(foo1.c, 100, 120)$ and the second is $(foo1.c, 105, 130)$, the intersection is 105-120 (16 lines), the union is 100-130 (31 lines), and the overlap is 16/31, or about 52 percent. The overlap is 1 if and only if the two fragments are identical.

A *clone pair* (or simply *clone*) is defined to be a triple (f_1, f_2, t), where $f1$ and $f2$ are fragments and t is a type (1, 2, or 3). The first fragment in a clone pair C may be referred to by $C.CF_1$ and the second by $C.CF_2$. Every clone pair C is assumed to be normalized (by swapping fragments if necessary) so that, if the two filenames are different, then the first precedes the second lexicographically, whereas, if the filenames are the same, then $C_1.CF_1.StartLine \leq C_2.CF_2.StartLine$, and, finally, if the filenames and start lines are the same, then $C_1.CF_2.EndLine \leq C_2.CF_2.EndLine$.

The degree to which two clones agree can be evaluated via the overlap of the two fragments as follows: The *good-value* between two clone pairs C_1 and C_2 is defined by

$$good(C_1, C_2) = min\{overlap(C_1.CF_1, C_2.CF_1), overlap(C_1.CF_2, C_2.CF_2)\}.$$

For example, if the first clone pair C_1 contains fragments $C_1.CF_1 = (foo1.c, 100, 120)$ and $C_1.CF_2 = (foo2.c, 200, 230)$, while the second clone pair C_2 contains fragments $C_2.CF_1 = (foo1.c, 105, 130)$ and $C_2.CF_2 = (foo2.c, 205, 235)$, then

$$good(C_1, C_2) = min(16/31, 26/36),$$

or about 52 percent.

Bellon et al. [11] defined a mapping procedure that maps candidates to references. It has the property that with respect to some value p, $0 < p < 1$, a candidate C is mapped to a reference R_1 in preference to a reference R_2 if $good(C, R_1) \geq p$ and $good(C, R_1) > good(C, R_2)$; however, if $good(C, R_1) < p$ or $good(C, R_1) = good(C, R_2)$, additional formulas are used to determine which mapping to prefer. Ultimately, a reference R was defined to have a good match with threshold p if and only if there is a candidate C such that C is mapped to R and $good(C, R) \geq p$. The standard threshold for "good" was chosen (arbitrarily) to be $p = 70\%$.

While this mapping procedure may seem reasonable on the surface, it has a flaw. A candidate could meet the 70 percent level with respect to two references but would be mapped to only one, and one reference would be regarded as not "found" by the tool.[2] Therefore, we do not use the definition of mapping in Bellon et al. [11]. Instead, we use the following straightforward definition:

Definition 1. *For each reference r and a set S of candidates, the recall of r is the largest value of $good(r, c)$ for any candidate c in S. For the set of references as a whole and a parameter p, the* p-recall *is the proportion of individual references for which the recall is at least p. The* good recall *(or simply* recall*) is the 70 percent recall, where 70 percent was chosen to be consistent with the choice in [11].*

To compute the rejection rate, we would ideally like to know for each seen-candidate whether Bellon generated a reference from it during oracling. Unfortunately, Bellon did not provide this data directly; he provided only a list of the candidates that were oracled and data for which candidates were mapped to which references (if any). If any reference was derived by oracling a seen-candidate and overlapped it, the seen-candidate would be mapped to it unless it were mapped to another reference instead; given the relatively small number of references, the probability of another reference overlapping the same seen-candidate is probably small and, hence, the number of such instances is probably also small. Hence, we can consider nonmapping as an indication of rejection in oracling.

Another possibility would be to consider a seen-candidate as rejected if it was unmapped or if it was mapped but the good-value compared to the reference mapped to was less than 70 percent. We will not include these mapped candidates in the set of rejected seen-candidates that we will analyze because the same factors

2. Bellon has since computed that this happened in only five instances.

TABLE 1
Recall and Rejection for Dup

	Recall			Rejection		
Language	Type 1	Type 2	Combined	Type 1	Type 2	Combined
C	51%	43%	48%	55%	35%	38%
Java	62%	49%	52%	18%	18%	18%

that caused the good-value to be less than 70 percent for recall would also apply for rejection and the discussion would therefore become repetitive. Instead, we focus on unmapped seen-candidates. For any particular tool, we adopt the following definition of rejection:

Definition 2. Rejection *is the number of unmapped seen-candidates divided by the number of seen-candidates.*

The recall and rejection for Dup in the experiment are shown in Table 1 for C and Java for Types 1 and 2 separately and combined. Dup did not generate Type 3 candidates for the experiment.

4 Recall and Factors Affecting It

This section explores the effect of different factors on Dup's recall. These include a filtering option used by Dup, the definitions of Type 2 versus p-match, and aspects of the oracling guidelines given in Section 3.2. To simplify the discussion, we discuss Type 1 and Type 2 together.

The method is to modify Dup and apply changed or new options to show that recall improves. Some of the new options, such as breaking up candidates based on nesting level or eliminating preprocessor statements from candidates, could be incorporated into a postprocessor that could be run on the output of any tool, potentially improving performance.

4.1 Filtering

When a user runs a clone-finding tool on a software system and it reports a large number of clones, the user may prefer to look only at the ones that are potentially most useful. P-matches (especially short ones) with many changed parameters often appear related by coincidence rather than through copying and modification. Hence, Dup, by default, discards p-matches in which the number of changed parameters is more than half the number of lines. In this experiment, with the emphasis on having a human determine whether candidates were appropriate as clones, it seemed likely that Bellon would reject candidates that would be of low interest to a developer and, therefore, Dup was run with this option. However, Bellon's oracling ignored numbers of changed parameters. Consequently, the filtering had a significant effect on recall.

For the 1,693 references from C projects, running Dup again without filtering increased good recall from 48 percent to 56 percent. The percentage of references with recall of zero decreased from 28 percent to 16 percent, while 100 percent recall increased from 27 percent to 30 percent. However, the total number of candidates increased from 33,278 to 405,385! Thus, increasing the recall by a factor of 1.18 was accompanied by an increase in the number of candidates by a factor of 12!

For the 1,793 references from Java projects, running Dup again without filtering increased good recall from 52 percent to 57 percent. As in the case of the C projects, the number of references with recall of zero decreased, this time from 18 percent to 11 percent, while 100 percent recall increased from 27 percent to 31 percent. Again, there was an increase in the number of candidates, from 30,398 to 87,924, a factor of 3 in exchange for a factor of 1.1 improvement in recall.

For this paper, it was not possible to measure how rejection changes when the additional candidates are generated, since that would require additional oracling by Bellon. One question is whether the large increase in the number of candidates matters. If the candidates are to be subjected to further processing via software, the number of candidates may not matter. However, if the candidates are to be examined by a person, the increased number of candidates could be overwhelming. Hence, whether the small increase in recall by eliminating filtering is worth the factors of 3-12 in number of candidates may depend on the intended application.

4.2 Type 2 versus p-Match

A p-match may be extendable to a longer Type 2 candidate because the Type 2 definition is less restrictive. For the experiment, Dup generated maximal p-matches. This section explores how the differing definitions affected the results.

As mentioned in Section 3, Dup originally worked by finding Type 2 candidates (limited to substitution of single parameters like identifiers or constants for single parameters) and postprocessing them to identify p-matches. In order to assess the effects of differing definitions on the results, a Type 2 option was put back into Dup.

The strategy for finding Type 2 candidates with Dup works as follows: First, modify the input by replacing each possible parameter by the same symbol, say P, e.g., transform $x = y$ to $P = P$; then, find exact clones. This trick was originally proposed by Kernighan [18]. Putting this in as an option for Dup required only a trivial change to the tokenizer. With respect to the original input, some of the resulting candidates may actually be Type 1 rather than merely Type 2. However, the candidates can be correctly classified via sorting them together with the original Type 1 candidates and then running the UNIX uniq command.

We can use the new option to evaluate the effect on recall of changing to the Type 2 definition. In fact, the effect on recall of changing from p-match to Type 2 was surprisingly small.

For the C projects, compared to unfiltered p-matches, the new Type 2 option increased good recall from 56 percent to 63 percent. The percentage of references with recall of zero was reduced from 16 percent to 12 percent, while 100 percent recall increased from 31 percent to 38 percent. The total number of candidates increased by 36 percent.

For the Java projects, the new Type 2 option actually decreased the number of references with recall of at least 70 percent from 1,029 (57.4 percent of all references) to 1,023 (57.1 percent of all references) out of 1,793 Java references. Apparently, some of the new matches were enough longer compared to the corresponding references that the good-value decreased. These instances were most likely due to Bellon's applying Guideline 2 (truncation based on nesting) in generating a reference. The percentage of references with recall of zero was reduced from 11 percent to 8 percent, while the number with 100 percent recall stayed exactly the

same (31 percent of the total). The total number of candidates increased by 27 percent.

4.3 Effect of Truncation Based on Nesting Level

Rather than having references represent maximal Type 2 clones, Bellon chose to require (by Guideline 2) that a reference cannot go from the initial level of nesting to a shallower one, e.g., from the inside of a loop through the end of the loop to following statements. As a result, Bellon truncated many seen-candidates in generating references. For a tool that does not apply this rule, the changed boundaries affected the good value of recall.

To evaluate this effect for Dup, an option was added to Dup to divide up candidates by nesting level based on braces. The strategy is to count opening and closing braces to determine nesting level and assign each line the nesting level of the start of the line. Adding this option required a trivial change to the lexical analyzer and the addition of code to break up a candidate by nesting level, if necessary, and to discard the ones below the threshold length.

This simple strategy is not accurate when the body of loops or `then` or `else` clauses consist of one statement without enclosing braces; additional single statements could also be nested deeper without braces. It also does not take into account other types of grouping, such as `case` statements, which do not require braces. However, most commonly, this strategy should not be off by more than one line, and it serves as a way to estimate the effect of the nesting guideline on recall.

For the C projects, adding the nesting option in addition to the Type 2 option increased the good recall from 63 percent to 69 percent. The number of references with 100 percent recall increased from 38 percent to 44 percent. However, the number of references with recall of zero also increased from 11.6 percent to 12.8 percent! Inspection of the 19 candidates for which the level changed from nonzero to zero shows that the nonzero values were from overlap of a reference with the tail ends of candidates that were truncated by the nesting option.

For the Java projects, the nesting option increased good recall from 57 percent to 65 percent. The number of references with 100 percent recall also increased from 31 percent to 41 percent. As in the case of the C systems, the number of references with recall of zero increased, this time from 8 percent to 11 percent due to truncation of parts of candidates that overlapped a reference.

In some cases, truncation caused a candidate to shrink below threshold length and be discarded, while in others, a single candidate could be broken up into multiple candidates. Overall, there was a net decrease of 3 percent in C candidates and a net decrease of 7 percent in Java candidates.

4.4 Candidates from Repetitive Regions

A repetitive region of code is a region where similar code fragments occur at least twice in succession. For example, the code may be of the form $A_1A_2...A_n$, where each A_i is a clone of each other A_j. In this section, we discuss possible ways to report clones in repetitive regions, the implications for the number of clones, how to identify repetitive regions, and the impact on recall.

```
public Dimension getPreferredSize(JComponent a) {
  Dimension returnValue =
    ((ComponentUI)(uis.elementAt(0))).getPreferredSize(a);
  for (int i = 1; i < uis.size(); i++) {
    ((ComponentUI)(uis.elementAt(i))).getPreferredSize(a);}
  return returnValue; }
```

Fig. 3. A Java method frequently cloned in j2skd1.4.0-javax-swing.

4.4.1 Ways of Reporting Clones in Repetitive Regions

If there is a pattern $A_1A_2...A_n$, where each A_i is a clone of each other A_j, there are various possibilities for reporting clones. One would be to report that there are n clones of A_1 in succession; however, there was no way to state this in the format of the experiment. A second strategy would be to report that the first $\lfloor n/2 \rfloor$ A_i's match the last $\lfloor n/2 \rfloor$ A_js; however, this approach does not capture the fact that the size of the repeating region is only $|A_i|$. Another possibility would be to report a clone for each A_i and A_j, amounting to $n(n-1)/2$ clones. A fourth is Dup's approach of reporting all maximal clones. For the above region, Dup would report $n-1$ clones, where the ith clone has one fragment that begins with A_1 and ends with A_i and another fragment that begins with A_{n-i+1} and ends with A_n. Note that the two fragments of a clone can overlap. These maximal $n-1$ clones imply all the matches between A_i and A_j for $i \neq j$.

Now, suppose there are two regions, one containing $A_1A_2...A_n$ and the other containing $B_1B_2...B_m$, $n <= m$, where each A_i matches (by Type 2) each B_i. One possibility would be to report only that $A_1A_2...A_n$ matches $B_1B_2...B_n$; however, this does not capture the fact that A_i and B_j match for $i \neq j$. Another possibility would be to report all the pairs A_i and B_j. As always, Dup reports all maximal clones with distinct pairings of lines. For Type 2 clones, if a maximal clone pairs A_1 with B_j, then it also pairs A_2 with $B_j + 1$, A_3 with $A_j + 2$, and so on, through A_r and B_s, where either $r = n$ or $s = m$. Hence, Dup reports $O(n+m)$ maximal clones, rather than $O(nm)$.

Finally, suppose that there are r regions with n repetitions each of the form A_i. Then, there are $O(r^2n)$ maximal Type 2 clones, or $O(r^2n^2)$ clones of the form A_i matching A_j. This situation can arise within a single file or between files, and the number of maximal clones can become very large.

For example, in the Java project j2sdk, there were 118 references consisting of matches between lines 44-143 of certain files. Since Bellon picked 2 percent of candidates at random to oracle, there should be about 50 times as many such clones in the entire set of candidates from all tools, or about 5,900 such clones. Most of the methods involved were Type 2 clones of the method shown in Fig. 3.

Moreover, for the same project, there were another 86 references that matched lines 1-143 of different Java files. Again, there should be about 50 times as many in the set of candidates from all tools, or about 4,300 such instances. These class files extended other class files and had similar method names. For example, the last method in each file was called `getAccessibleChild`.

4.4.2 Estimating the Amount of Repetitive Code

The number and size of repetitive regions can be estimated by looking at the Dup candidates whose form indicates repetition, that is, at candidates in which the two fragments either touch or overlap, as in the $A_1A_2...A_n$ example above.

By defining the region covered by such a clone as repetitive, and merging repetitive regions that overlap or touch, we can estimate the amount of repetitive code. The following analysis is based on the candidates produced by Dup with the options for Type 2, nesting, and ignoring preprocessor statements (in the case of C code).

For the C projects, this analysis estimates that there are 1,507 repetitive regions as determined by 27,174 overlapping or touching Type 1 or 2 candidates. These regions include 45,367 noncommentary source lines, 12 percent of the total, with an average size of 30 lines. For the Java projects, this analysis estimates that there are 1,360 regions of repetitive code, as determined by 6,833 overlapping or touching Type 1 or 2 candidates. These regions contained 37,991 noncommentary source lines, 17 percent of the total, and had an average size of 28 lines.

4.4.3 The Effect of Repetitive Regions on Recall

This section studies two ways in which repetitive regions could have affected recall: candidates including preprocessor statements and the style in which candidates are reported in repetitive regions.

In C, `include` statements often appear in blocks at the start of each file. If these are allowed to appear in candidates, then these form repetitive regions with the blowup in number of candidates that was described earlier. Bellon, however, allowed a reference to contain preprocessor statements only if a candidate fragment was the whole file. For a tool that allows preprocessor statements in candidates, this would affect recall mainly by the impact of Bellon's removing them in generating references; removing the preprocessor statements from candidates should improve the goodness of recall for some references.

Since Dup allowed preprocessor statements to appear in candidates, running Dup without filtering but with the Type 2 and nesting options generated many candidates from sequences of `include` statements. Rerunning Dup with an additional option to exclude preprocessor statements improved the good recall by only 0.4 percent. However, the number of C candidates was reduced by 46 percent.

Different tools used different reporting styles for repetitive regions. To the extent that these styles were preserved in references, the different reporting styles affected all the tools. In fact, Bellon asserted that he left references in repetitive regions in the style in which they were reported (Guideline 5). However, inspection showed that he never generated a reference with overlapping fragments. Besides Dup, two other tools in the experiment generated overlapping fragments.

The following analysis of the references and the candidates within or between repetitive regions provides an estimate of how many references failed to have recall of at least 70 percent because of the different styles of reporting candidates in repetitive regions. For C, of the 519 references with recall below 70 percent, 72 are at least 70 percent covered by candidates that are within or between repetitive regions, and only three of these are not 100 percent covered. Of the Java references below the 70 percent threshold for recall, 146 are at least 70 percent covered by candidates that are within or between repetitive areas, and only two of these are not 100 percent covered. These 72 C references and 146 Java references are instances where the correspondence was found, but Dup's reporting method differed from the reference.

In summary, the effect of preprocessor statements on recall was negligible, but references that have recall less than 70 percent but are at least 70 percent covered by candidates within or between repetitive areas account for 4 percent of all references in C and 8 percent in Java, apparently due to different reporting styles for candidates in repetitive regions.

4.5 Other Factors

Finally, we account for the remaining references (447 for C, or 26 percent, and 477 for Java, or 22 percent) whose recall is less than 70 percent and which are not at least 70 percent covered by candidates within or between repetitive regions, even for candidates generated using the new Dup options. Among these are a number of references with a recall of zero: 221 (13 percent) out of 1,693 references for the C systems and 199 (11 percent) out of 1,793 references for the Java systems.

Inspection of some of these references with recall of zero reveals a number of reasons that a reference was not found as a candidate by Dup:

1. reformatting ("normalization") by Bellon that removed a line break before a closing brace in one fragment but not in the other, where a comment followed the brace,
2. original layout differences, e.g., when one fragment has line breaks where the other does not, whereas Dup requires that line breaks be the same,
3. fragments that have fewer than six lines of ncsl, as a result of Bellon's allowing a commentary line to count toward the six, in Guideline 1 above, whereas Dup has a threshold of six ncsl,
4. references in which type keywords are substituted for others, or in Java references, where the `super` keyword was substituted for, whereas Dup does not treat keywords as parameters,
5. fragments in which an expression is substituted for another, as in `args->string[0]` substituted for `name` as in Guideline 3, rather than a single parameter for another, as required by Dup,
6. a few references that appear to have been misoracled by Bellon, e.g., a Type 3 reference (with an extra line inserted in the middle in one fragment) categorized as Type 2, and a reference in which the initial line was in a `then` clause (and, hence, at a deeper level of nesting) in one but not the other.

While this list of factors was obtained by eye from references with recall of zero, they would also affect the references for which the recall was greater than zero but less than 70 percent.

For reasons 1 and 2, the problem could be eliminated by switching Dup to an ordinary tokenizer rather than using its current semi-line-based approach. For reason 3, the problem could be eliminated by counting lines based on actual lines rather than noncommentary source lines. For reason 4, the effect could be partially reduced by allowing types to be parameters, but Dup could not cope with substituting a multiple-word parameter type for a single-word parameter type. Dup could not handle reason 5. Substitution of multiple tokens for single parameters is

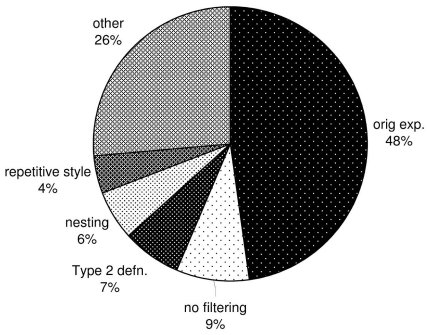

Fig. 4. Factors affecting recall for C.

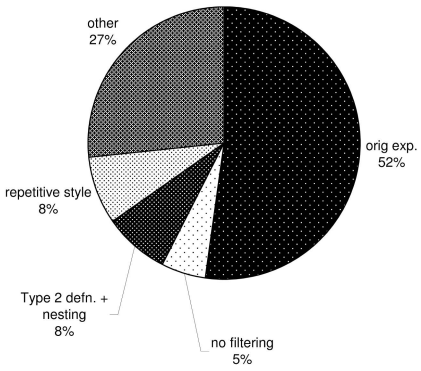

Fig. 5. Factors affecting recall for Java.

fundamentally outside the scope of algorithms based on p-suffix trees.

4.6 Summary of Factors Affecting Recall

The pie charts in Figs. 4 and 5 summarize the data for the factors affecting Dup's recall for the C and Java projects. The biggest surprise was that switching from p-matches in the experiment to Type 2 made a relatively small difference: only 7 percent for C and no difference at all for Java except when combined with the nesting option.

If the experiment were run again, the new or changed options in Dup could be used to improve recall for Dup from 48 percent to 70 percent for C and from 52 percent to 65 percent for Java. However, the increase in the number of candidates might also mean an increase in rejection, which cannot be measured directly without additional oracling. The next section analyzes factors affecting Dup's rejection in the experiment. In fact, the options of nesting, Type 2, and excluding preprocessor statements are shown to decrease rejection for the seen-candidates from the experiment.

While code to break up candidates based on nesting was put into Dup for this experiment, a better job could be done by a postprocessor with a parser. This postprocessor could be run on the output from any tool, with potential improvement to recall. In addition, a language-based option, such as excluding preprocessor statements, could also be put into a postprocessor.

5 Factors Affecting Rejection

In this section, we investigate why oracling failed to generate references from some of the seen-candidates. Recall that Bellon did not record for each seen-candidate whether he generated a reference from it and, hence, we use nonmapping of a seen-candidate to a reference as our criterion for whether he rejected a candidate as a clone.

The approach is to rerun Dup with new options and to show that fewer of the unmapped seen-candidates are generated. The factors considered are truncation based on nesting Guideline 2 and certain language features. In some instances, Bellon justifiably rejected candidates that appeared of low interest for a user, although he appeared to be inconsistent about his criteria. These cases give some feel for marginal situations that occurred in oracling and illuminate the difficulty of automating the evaluation of the quality of a candidate. As in the case of recall, a postprocessor could incorporate some of the Dup modifications that reduce rejection (and increase recall) as well as checking for other language features shown to be relevant, and would have the potential of improving performance for any tool.

In the experiment, Dup reported 252 C candidates that were seen but not mapped. For this paper, to analyze why these candidates were not mapped, Dup was run again using the nesting option and the Type 2 definition, which, in some cases, generates longer candidates. Of the 252 seen-candidates, 65 did not overlap any of the candidates generated in the new run. Apparently, in these instances, if Bellon had tried to generate a reference by applying Guideline 4 to expand the candidate to a maximal Type 2 candidate and then truncated it by nesting using Guideline 2, the resulting reference would have been shorter than the six-line threshold. Hence, the main reason for rejecting these seen-candidates is that Dup did not prune candidates based on nesting before applying the six-line threshold.

Of the 187 C seen-candidates that were still generated with the nesting option and Type 2, 26 did not overlap any candidate generated when Dup was run again with an additional change to ignore preprocessor statements. Apparently, removing preprocessor statements reduced these candidates to below six lines.

Inspection of the remaining unmapped C seen-candidates revealed 69 that were fragments of table initializations. Another 59 fell into two classes that cannot be completely distinguished without a clean definition of proper nesting:

- The new Dup option of truncation when nesting level decreases from the initial level was inadequate for the particular situation, and, if properly truncated, the remaining code would be too short. In particular, the simple method of counting braces and looking at the nesting level of the start of the line does not handle two types of instances properly: `case` statements, which do not need braces, and lines consisting of `} else if {` , where the nesting level changes within the line.
- After a possible truncation, the form of the fragments was not suitable as a clone. Common examples were the start of a procedure, including the type declaration and the first few lines of the body, or the start of a `case` statement.

The above 59 unmapped seen-candidates will be termed instances of rejection due to improper nesting form.

The rejection of the remaining C seen-candidates appeared to be for miscellaneous reasons. Twenty-two were repetitive regions consisting of sequences of two or

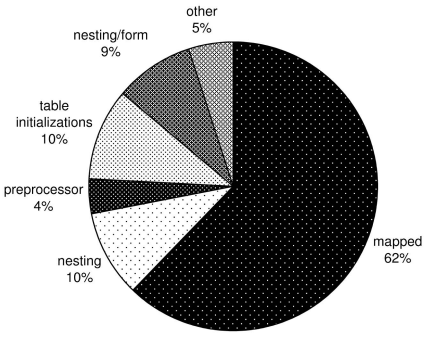

Fig. 6. Factors affecting rejection for C.

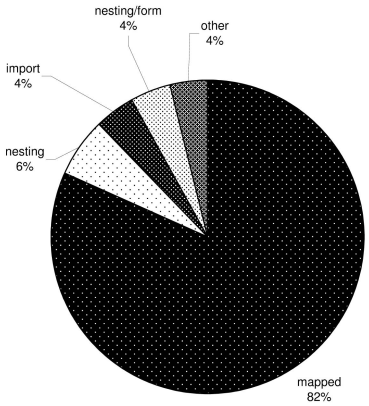

Fig. 7. Factors affecting rejection for Java.

more short items such as `case` statements (with associated code), multiline code patterns, or `fprintf` statements. (Dup applies the six-line minimum to the matching sequences, not the individual items.) Two were code intermixed with `ifdef` preprocessor statements. Six were matches between the parameters of procedure calls that were spread out over multiple lines. Three were miscellaneous sequences of lines of code.

The rejected candidates just described fall into categories that are perfectly reasonable to disallow if a clone is to represent a syntactic or semantic entity and the threshold size is to apply to individual items in a sequence. However, Bellon was not consistent about rejecting candidates in these categories. He allowed 83 references from Dup that involved repeated patterns, each of length less than six lines; these were mainly sequences of calls to `fopen` or other functions, but also included sequences of `case` statements and structure initializations. Also among the references are fragments consisting of procedure declaration followed by a few lines of type declarations in the body. It is not obvious what other criteria might distinguish the accepted references from the rejected candidates. Since Bellon did not specify a formal set of guidelines for oracling and oracled by eye rather than via software, these instances may simply fall into a gray area of human inconsistency or inaccuracy.

Dup's choice of reporting repetitive regions via candidates in which one fragment overlaps the other was not a big factor in rejection of the short repeated patterns. In general, Bellon did not simply reject candidates of this form in oracling but rather extracted references without overlaps where possible. There were 10 seen-candidates of this form that were mapped. Among the unmapped seen-candidates, there were only 11 with an overlap, and of these, all but two were parts of table initializations, which were usually rejected anyway.

Repeating this analysis for the 112 Java candidates that were seen but not mapped shows 38 that were not generated using the nesting option, even with the Type 2 option. Inspection of the remaining 74 Java candidates revealed that they fell in various categories:

- `Import` statements accounted for 25; Java `import` statements are similar to C `include` statements in occurring in blocks at the start of files.
- Repetitive regions consisting of sequences of small items (case statements or methods, each shorter than six lines) accounted for 14.
- Too-simple calculation of nesting accounts for 26 that, when truncated in a more sophisticated manner at the beginning or end, would have been less than six lines. Of these, 10 needed truncation of method fragments at the end, six required truncating for `case` statements or lines of the form `} else if {` (where nesting level changed in the middle of the line), and 10 needed truncation at the end because they ended in the middle of an expression or nested within an `if` statement.
- In five instances, after truncating by nesting, the candidate contained whole methods consisting of one line of declaration and five lines of body; Bellon presumably decided in these instances that he required a six-line body.
- One was part of a method invocation that spread over several lines.
- One was part of a table initialization.
- One had overlapping fragments, and with proper truncation of case statements, would not include six-line nonoverlapping matching fragments.

As in the case of C, some of these instances of rejection (notably the repetitive regions consisting of sequences of small items) appear inconsistent with other references that were accepted.

The percent by which the above factors affected rejection are summarized in Figs. 6 and 7. The "nesting" category refers to the simple nesting option as implemented in Dup. The "nesting form" category refers to the two categories mentioned earlier, one where the Dup nesting option was not sufficiently precise and the other where Bellon apparently felt that the form of the candidates did not warrant a reference.

Without additional oracling, there is no way of determining whether the overall rejection rate would be improved for the new total set of candidates compared to the original, but it seems likely that it would. A postprocessor could incorporate breaking up candidates based on nesting (and could do it properly based on parsing) as well as checking for features like preprocessing statements, table initializations, and repetitive regions consisting of sequences of small items. Such a postprocessor could be run on any of the tools, potentially improving performance.

6 The Experiment in Retrospect

The experiment described in [11] was a good exercise in trying to design an evaluation method for comparing tools based on very different techniques and producing different kinds of output. Bellon is to be commended for his

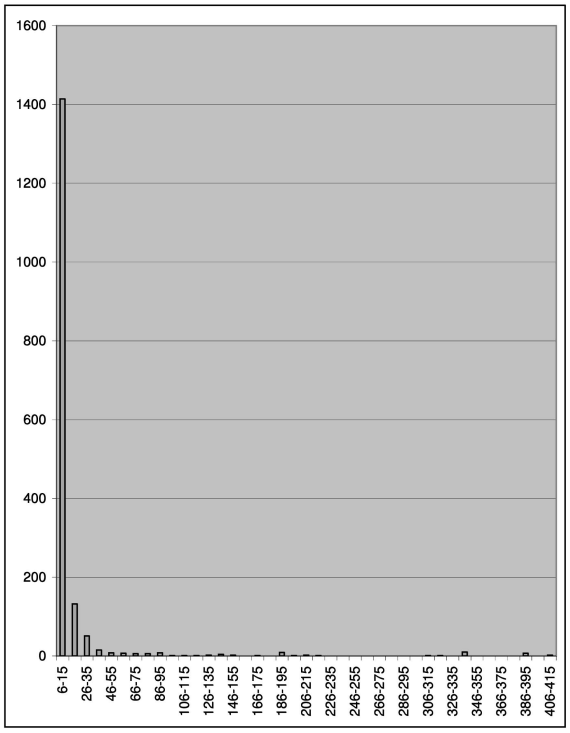

Fig. 8. Size distribution of references in C.

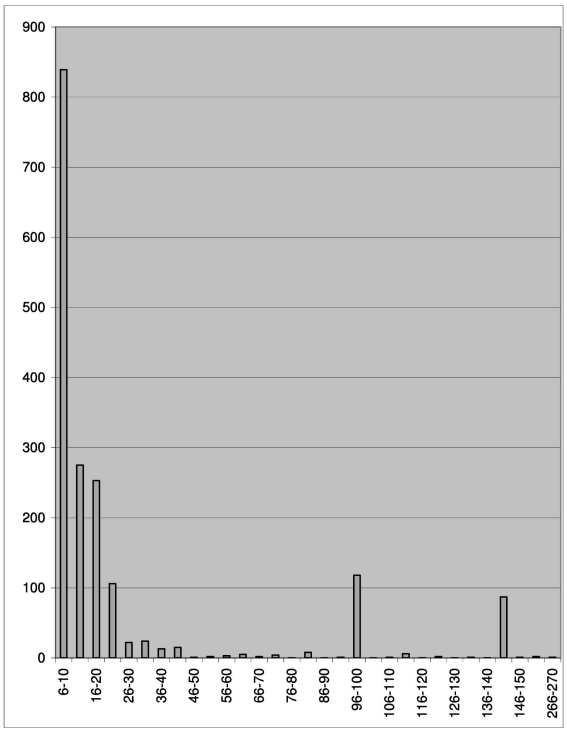

Fig. 9. Size distribution of references in Java.

ingenuity in devising an experimental method that accommodated all the tools and enabled a comparison between them. Nevertheless, an overall comparison of the tools (e.g., to declare a "winner") is difficult to provide; for one thing, it would not be fair to combine the results for all the input systems because some of the tools could not process all the systems for technical reasons, and for another, the tools varied as to which of Types 1, 2, and 3 they generated.

Much of the effort in designing the experiment went into sorting out how to deal with the differences between tools. The design was intentionally imprecise in order to allow different tools to participate. However, the types of the clones should have been more clearly defined. It was unfortunate that he used the term "parameterized" to mean Type 2 (as defined in retrospect) rather than to mean parameterized matches as accepted in the literature. His retrospective definition of Type 2 was still vague as to what kinds of substitutions were allowed for parameters. Type 3 was vaguely defined, as well.

Bellon did not announce in advance how he would oracle candidates. In retrospect, his oracling was mainly systematic and describable by the guidelines presented in Section 3.2, although some inconsistencies appeared in the above analysis. He seemed mostly to take textual and syntactic features into account and did not seem to make judgments based on other aspects of the quality of the candidates (unless those are reflected in the above-mentioned inconsistencies). Because the quantity of candidates to look at was so large, he did not have the leisure to spend a lot of time examining each candidate to look at semantics, for example.

In contrast, it seemed reasonable to expect that Bellon would reject some candidates that were textually and syntactically similar on the grounds that they would not be useful as clones. This expectation influenced the choices made for Dup (particularly with regard to filtering) and possibly for other tools when providing data for the experiment.

One might wonder whether the particular systems that comprised the input contained peculiarities that influenced the results. In fact, the reference sizes have a very uneven distribution, as shown in Figs. 8 and 9. The references of length 6-10 greatly dominate references of other sizes. A difference of one or two lines in candidate boundaries has a larger impact on good value for short candidates than for long candidates. For Java, the reference sizes show a couple of bumps in the ranges 96-100 and 141-145; these references were described earlier in the discussion of quadratic blowup in the number of clones due to similar repetitive regions in different files. In addition, since the individual regions contained many similar six-line methods, tools that reported pairs of methods as candidates rather than pairs of sequences of methods would have reported many six-line candidates. The references resulting from these would be included in the large number of references of length 6-10. These long repetitive code sections may be viewed as all being derived from one example of code structure that had a great impact on the results. While any system studied might have some repetitive code, it is hard to know if the particular systems analyzed were typical.

Earlier, we estimated that 12 percent of the C systems and 17 percent of the Java systems consisted of repetitive regions. However, when Dup was run to estimate the size of repetitive regions, 82 percent of C candidates and 62 percent of all Java candidates were within or between repetitive regions. If the figures were similar for the other tools (and they might even be higher since some other ways of reporting clones in repetitive regions lead to larger numbers of clones being reported), then the repetitive

regions had an effect on the results out of proportion to their size.

The candidates represent a correspondence between the (noncommentary) lines in two code fragments. Bellon's method of mapping a candidate to a reference depended only on overlaps between the reference fragments and the candidate fragments. Sometimes a Dup candidate was mapped to a reference that matched up completely different pairs of lines. Such anomalies were unavoidable because the experiment methodology specified candidates and references only in terms of beginning and ending line numbers.

The analysis in this paper suggests that knowing in advance how the oracling would be done would allow Dup, and probably other tools, to improve performance by means of small modifications and extra options without changing the core algorithms. This makes it hard to know just what the experiment actually measured in comparing tools. The next section has some suggestions on how to modify the experimental method to make the results more meaningful.

7 Conclusions and Areas for Future Work

Compared to previous work on finding duplication in source code, the experiment described in [11] included three novel elements:

- A data set of clone candidates was produced by an assortment of techniques, which a tool owner could not do independently.
- A person oracled a substantial number of these to produce a set of clone references.
- A methodology was developed to determine which candidates corresponded to which references, even though the boundaries might be different.

The inclusion of various tools was of huge importance in identifying what each tool was missing that might be considered interesting as a clone. While the oracling represents just one individual's notion of what was appropriate as a clone, and it would certainly be preferable to get the judgment of many individuals, the oracling still represented human input. The methodology for finding correspondences between candidates and references worked well and should be useful for future experiments.

The experiment studied three types of clones, but at least six types could have been distinguished: exact clones, p-matches, Type 2 limited to substitution of individual parameters, Type 2 allowing substitution of expressions for parameters, Type 3 limited to allowing small insertions and deletions within a line (as in changing the number of arguments to a function), and Type 3 allowing insertions, changes, and deletions of whole lines. These are, however, loose descriptions. It would be desirable to create a careful set of definitions for use in future studies.

In comparing tools, it would be desirable to reduce the impact of the tool differences that depend on choices that are not related to the core tool algorithms, such as whether to take nesting into account or how to treat various language features. While Bellon actually looked at 2 percent of the candidates, his oracling was rather systematic and was based mainly on textual and syntactic features. The Guidelines and the modifications to Dup for this paper showed that Bellon's oracling could be largely automated. His guidelines for redefining the boundaries of candidates to produce references, along with certain other oracling choices (such as whether to allow clones to contain preprocessor statements or table initializations) could be incorporated into a postprocessor that could be run on candidates before oracling or evaluation.

In fact, if it is difficult for a future experiment to find one or more people to oracle large numbers of candidates to create references, an alternative would be to have people define sets of preferences for oracling that could be incorporated into postprocessors that would be run on candidates, and then the tools would be compared on the set of postprocessed candidates. By requiring a smaller amount of time, this approach might enable an experiment to have multiple people express their preference for what constitutes a clone, rather than just the one oracle that our experiment had.

Furthermore, the automatability of much of Bellon's oracling suggests that two types of tool would be useful for clone-related software engineering. *Clone-finders* would find clone candidates, and *clone-massagers* would redefine the candidate boundaries and possibly discard some candidates. Different people might have different preferences for clone-massagers, and different boundaries might be useful for different purposes, e.g., according to whether they are to be used directly by a person or subjected to further processing. For example, shrinking clones based on nesting might be desirable for application to refactoring but not for other purposes, such as using Type 2 candidates to find Type 3 candidates. (A Type 3 candidate can be viewed as a Type 2 candidate in which additional small changes have been made. The additional small changes would divide the original Type 2 candidate into two or more smaller Type 2 candidates separated by the small changes. To reverse-engineer this process, some Type 3 candidates can be found by identifying successive Type 2 candidates separated by small amounts of code. In this case, it would be undesirable to shrink the Type 2 candidates based on nesting first.) It would be useful to formalize a set of applications and create clone-massagers for them.

How to handle repetitive regions is an important area for further research. One issue is the best way to report candidates in these regions. As described earlier, different tools use different methods. Dup's method of reporting maximal (possibly overlapping) matches reports complete information about correspondences between fragments and can be used to generate the other types of reporting methods for repetitive regions. Hence, if other tools could use Dup's method, the final reporting method could be incorporated into clone-massagers.

It would be good to define additional evaluation methods that would keep repetitive regions and large numbers of copies of particular structures from dominating the results to such an extent that performance in other situations is hard to evaluate. The simplest approach would be to give results separately for repetitive and nonrepetitive regions of code. It might also be possible for the evaluation to use a weighting function based on the number of candidates or references overlapping on each line of code, with the goal of giving every line of code equal weight.

Another area for future research is measuring the quality of candidates. For a person who wants to examine candidates in order to improve code, the sheer quantity of output from these tools is a huge deterrent. Ranking the

candidates by usefulness or weighting them by some quality measure could be very helpful. Length is an obvious factor, since very long candidates are likely not to arise by chance, and structure or included language features may also be important. The usefulness of a particular candidate might depend on the desired application. Clone-massagers could be useful for this task as well.

The format of this experiment provided only for a one-line description of each clone. The various tools differ in their usual output format. Dup, for example, also reports for each p-match how the parameters were renamed and has an option for evaluating the location and amount of duplication. Such additional features from various tools would be hard to accommodate in an experiment like this one, however.

ACKNOWLEDGMENTS

The author would like to thank all the participants for the considerable effort they put into the experiment. This paper was supported in part by Bell Laboratories, Alcatel-Lucent.

REFERENCES

[1] B.S. Baker, "A Theory of Parameterized Pattern Matching: Algorithms and Applications," *Proc. 25th ACM Symp. Theory of Computing,* pp. 71-80, May 1993.
[2] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second IEEE Working Conf. Reverse Eng.,* pp. 86-95, July 1995.
[3] B.S. Baker, "Parameterized Pattern Matching: Algorithms and Applications," *J. Computer and System Sciences,* vol. 52, no. 1, pp. 28-42, Feb. 1996.
[4] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM J. Computing,* vol. 26, no. 5, pp. 1343-1362, Oct. 1997.
[5] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, 1998.
[6] 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.
[7] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth Working Conf. Reverse Eng. (WCRE '01),* pp. 301-309, 2001.
[8] B. Lague, D. Proulx, J. Mayrand, E. 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.
[9] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance,* pp. 244-253, 1996.
[10] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance (ICSM '99),* pp. 109-118, 1999.
[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Trans. Software Eng.,* to appear.
[12] S. Bellon, "Detection of Software Clones," http://www.bauhaus-stuttgart.de/clones, 2004.
[13] E. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM,* vol. 23, no. 2, pp. 262-272, 1976.
[14] M. Crochemore and W. Rytter, *Jewels of Stringology.* World Scientific, 2003.
[15] A. Amir, M. Farach, and S. Muthukrishnan, "Alphabet Dependence in Parameterized Matching," *Information Processing Letters,* vol. 49, pp. 111-115, 1994.
[16] R.M. Idury and A.A. Schaffer, "Multiple Matching of Parameterized Patterns," *Proc. Fifth Ann. Symp. Combinatorial Pattern Matching (CPM '94),* M. Crochemore and D. Gusfield, eds., pp. 226-239, June 1994.
[17] S.R. Kosaraju, "Faster Algorithms for the Construction of Parameterized Suffix Trees," *Proc. 36th Ann. Symp. Foundations of Computer Science (FOCS '95),* pp. 631-639, 1995.
[18] B. Kernighan, personal comm., 1991.

Brenda S. Baker received the AB, MS, and PhD degrees in applied mathematics from Harvard University. After many years in the Computing Sciences Research Center at Bell Laboratories (through its incarnations as part of the Bell System, then AT&T, and then Lucent Technologies), she retired as a Distinguished Member of the Technical Staff and is now a consultant. Her early career also included interludes as instructor and Vinton-Hayes Research Fellow in the Division of Engineering and Applied Physics at Harvard University, visiting lecturer in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley, and assistant professor in the Department of Computer and Communication Sciences at the University of Michigan, Ann Arbor. Her current research is on algorithms and software tools to solve real-world problems. In algorithms, her interests are in string pattern matching, combinatorial algorithms, and approximation algorithms for NP-hard problems. Application areas she has contributed to include code analysis, Internet infrastructure, and robotics.