To appear in Proceedings of the 10th IEEE Working Conference on Reverse Engineering, *Nov 13–16, 2003, Victoria, Canada*

Problems Creating Task-relevant Clone Detection Reference Data

Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia

Software Research Laboratory
Center for Advanced Computer Science
University of Louisiana at Lafayette
E-mail: walenste@ieee.org

Abstract

One prevalent method for evaluating the results of automated software analysis tools is to compare the tools' output to the judgment of human experts. This evaluation strategy is commonly assumed in the field of software clone detector research. We report our experiences from a study using several human judges who tried to establish "reference sets" of function clones for several medium-sized software systems written in C. The study employed multiple judges and followed a process typical for inter-coder reliability assurance wherein coders discussed classification discrepancies until consensus is reached. A high level of disagreement was found for reference sets made specifically for reengineering task contexts. The results, although preliminary, raise questions about limitations of prior clone detector evaluations and other similar tool evaluations. Implications are drawn for future work on reference data generation, tool evaluations, and benchmarking efforts.

1. Introduction

A "software clone" is a copy or near-copy of a portion of code. Clones are frequently introduced by code scavenging, that is, by copying existing code and modifying it. Finding clones in software systems is important in many maintenance, reengineering, and program understanding contexts. In response, several automated clone detectors have been proposed (e.g., Baxter *et. al* [2], Kamiya *et. al* [7]).

Unfortunately, a precise definition of what differentiates a clone from a non-clone is lacking. This can present problems for evaluating clone detectors. The evaluators must decide for themselves what the "correct" list of clones should be. Because different evaluators may decide on different definitions, this can introduce variability and arbitrariness, and the various results may be well nigh incomparable [3].

One evaluation technique used to combat these difficulties is to evaluate automated tool results against a "gold standard", "reference set", or "reference corpus" data generated by humans (i.e., experts). For instance, Girard *et. al* [6] proposed a method for evaluating software clustering techniques against reference clusters generated by human experts. Human judges are thus treated as oracles. Then, perhaps, the vagueness in the criteria used to classify clones causes no harm because the oracle resolves the uncertainties. Yet even experts may disagree, perhaps especially when performing vaguely-defined categorization task—they may not be *reliable* oracles. In the study by Girard *et. al* [6] the judges were thought to be sufficiently reliable [10], however little is known about the reliability of judges in software engineering. Prior clone detection studies by Bellon [3], Burd *et. al* [4], and Kontogiannis [9], utilized human-generated reference sets, however none of these studies reported judge reliability.

This paper reports on a study which highlights problems observed in generating reference corpora for clone detection. The key step in the study was having several of us serve as clone detection oracles by classifying a selection of candidate clones into clone and non-clone categories. Our expectation was that reasonably informed researchers on software clones—such as ourselves—would able to agree in most cases whether some particular pair of code fragments were clones or not. What we found was an unexpected level of disagreement. For instance, for one specific set of 317 candidate clones, our three judges independently classified only 5 of them identically. Clearly, consensus is not automatic, and some care may be needed when generating reference data. The study was exploratory in nature, so the results are, at best, only suggestive. Nonetheless, the outcome raises red flags about prior clone detection research, and creates implications for ongoing benchmarking efforts and for many other tool evaluation techniques that rely on reference data generated by human judges.

Section 2 provides relevant background information on clone detection and motivates the study. The study and its results are described in Sections 4-8. Section 9 presents our tentative conclusions and implications for future research.

2. Background and motivation

In the absence of clear, precise, and standardized set of output requirements, it is difficult to evaluate automated reverse engineering tools such as clone detectors. This section outlines some of these problems so as to motivate the research questions being investigated in this paper.

2.1. Definitional vagueness

Kamiya *et. al* [7] describe "code clones" as "a code portion in source files that is identical or similar to another" [7, pg. 654]. What is meant by "similar" is unspecified. This level of definitional vagueness is typical within clone detection publications. Burd *et. al* [4], for instance, say that "a clone is recognized to be where a second or more occurrences of source code is repeated with or without minor modifications." [4, pg. 36]. It should be clear that when building a clone detector, knowing what "similar" and "minor" mean in the above definitions is positively crucial to success.

Attempts have been made to add precision to the definition by separating out various classes of clones. For instance, Mayrand *et. al* [14] define an ordinal scale of 8 different types of clones, of which some have simple, crisp definitions. For instance, their *DistinctName* identifies a well-defined category of clones which differ only by identifiers used. Unfortunately, such sub-categories can be seen to merely peel away those cases of similarity that are easily formalized. Mayrand *et. al*, for example, still include a category called *SimilarExpression*, which is intended to identify clones with expressions that differ but yet are still "similar". Other studies, such as the one by Kontogiannis [9], beg the definition question by assuming an approximated definition of "similar" based on (arbitrary) thresholds for allowable variation along various dimensions of differences. Attempts to skirt the problem by combining multiple detector result sets automatically (e.g., Mitchell *et. al* [15], Kontogiannis [9]) may be pragmatically helpful for tool evaluators, but still leaves open the question of how well the results match what human judges would decide.

2.2. Task-irrelevance

What is considered to be a *relevant* clone may depend on potentially many contextual factors, such as the system's coding style, and the tool users' task context. Clone detectors might be used for several purposes: finding code to refactor in perfective maintenance [4], finding duplicates to remove to reduce code size for mobile devices, and so on. The desired output for clone detectors in each case may differ. For example, it may be desirable in code compaction to refactor small, single-line clones even if doing so increases coupling, increases calling overhead, and decreases readability. In other perfective maintenance contexts these clones are not a concern. Contextual factors also appear in cases where portions of systems are created by a code generator such as `yacc`: the generated code might be otherwise indistinguishable from hand-generated verbatim clones, however normally the potential clones are not a refactoring concern.

Of course, one may still argue that even if the *usefulness* of clone detector results are contextually dependent, this in no way affects the actual *definition* of a clone: a clone is a clone, whether it is useful to identify it or not. This stance may provide little consolation to tool evaluators. To them, the issue is not finding clones *per se*, but finding *relevant* clones so that they evaluate the accuracy and hence efficacy of their tools. A context-independent standard may fail them in this purpose. For instance, a detector that finds 98% of the reference clones might fail to find any of the 2% of these that *actually matter* to the maintainer, making the detector in realty worse than one finding only 3% of the reference clones but 100% of the material ones. Thus a context-independent definition for cloneship may be academically justifiable, yet is in danger of serving little practical purpose.

One possible response to task irrelevance is to distinguish between reference sets for clones generally, and benchmarking reference sets for particular application contexts. Then several benchmarking reference data sets could be described as selections or filters on the general, task-independent reference set. That is, benchmarking sets would be subsets of the full reference clone set. Each such benchmarking subset would be attuned to a particular context. Doing so may help solve some contextual issues, however with each reference set one still faces the two other problems identified in this section.

2.3. Human judgement limitations

Human-generated reference sets have previously been proposed as a way of establishing performance standards in the absence of a precise definition of correctness. The basic idea has been used in automated reverse engineering and maintenance tools (e.g., Girard *et. al* [6]), but reference corpora are used widely, and can be found wherever vaguely- or subjectively-defined problems occur, such as in market research, medicine, linguistics, information retrieval, and image analysis.

It has long been known that there are several potential pitfalls with human-generated reference sets (e.g., Rust *et. al* [16], Landis *et. al* [13]). First, there is a threat that different observers will not agree on the same answer. The level to which the judges agree is generally called the "reliability" of the judgments. Reliability issues can call

into question the meaning and validity of human-generated reference data. Second, there is a threat that the particular definitions and instruments given to the judges may influence their results or the reliability of their results [10].

In other fields, techniques have been developed to measure and improve reliability between judges, and the importance of including reliability measures for subjective measures has been emphasized [8]. Although some software engineering studies can be found that report reliability (e.g., Girard *et. al* [6]), we know of none for clone detection, and we know of no studies in the field investigating reliability specifically.

3. Research questions

The problem of finding clones appears to fall into a class of problems with no precise definition of suitable output, and no specific and universal task context. The reliability of human-generated clone detector reference data is also unsettled. In order to investigate these issues we ran an exploratory study directed at generating a collection of task-specific references sets. It would also have been possible to seek out a task-independent reference set, but we wished to set the performance bar high specifically for clone detection for reengineering purposes. Our particular focus was on a reference set for *function clones*. Function clones are simply clones that are restricted to refer to entire functions or procedures.[1] This reference data would be useful for evaluating clone detectors which match only whole functions, such as the one by Mayrand *et. al* [14].

Although the generated reference data might be useful in itself, the contribution of this paper revolves around our struggles creating this reference set. At the outset, we had anticipated several potential problems in generating a trustworthy function clone reference set. These can be summarized by listing them as unsolved questions:

1. **What clone classification criteria would be appropriate for benchmarking clone detectors?** Burd *et. al* [4] had previously generated reengineering-specific reference data, but they did not publish specific guidelines detailing which clone candidates were considered "useful" in such a context. We had few explicit hypotheses other than a general notion that the clones should be refactorable, so we were engaged primarily in bottom-up empiricism on this point.

2. **Is inter-rater reliability a problem?** Given that reliability is an issue elsewhere, we proceeded as if we were testing the null hypothesis that judges were reliable and consistent.

3. **What is the role of practice effects?** Human oracles are being used because a precise definition of the desired results are unknown. By the very nature of this use of human judgment, the judges can, at best, be given only general rules and guidelines about how to classify the candidate clones. It is reasonable to expect that judges may classify differently after some practice in examining clones from various systems and contexts. How important is this learning effect? What guidelines or coaching could be given to future judges to reduce practice effects?

4. **Do system specifics make a difference?** The particulars of a system might make an impact on the reliability of judges and on the suitability or completeness of clone classification guidelines.

[1] The term "function clone" is already an accepted term so we shall adopt the term even if we mean either procedures or functions.

4. Study outline

We ran an exploratory study in order to investigate the research questions of Section 3. Our investigations are presented below in four-phases. All four phases employed a basic design for empirical trials: (1) a tractable sample of clone candidates is provided to multiple judges, (2) the judges independently classify the candidates, and then (3) a joint consensus-building session is conducted and all discrepancies between judgments are resolved. This basic flow of each run is illustrated in Figure 1. Figure 1 also shows the manipulable input variables: (1) the subject software systems, (2) a δ value used to select candidate clones, (3) the selection of judges, and (4) the instructions and clone classification guidelines given to judges. The output variables consisted of: (1) a reference set, and (2) measures of inter-rater reliability.

Each phase explored the research questions outlined above by manipulating input variables in a various ways and then observing the effects on the output of the process. Results from one phase were fed into the next. Details of each phase are described in separate sections below. Phase I manipulated value of δ (candidate set selection), after which it was fixed. Experience and feedback from performing the individual and group consensus-making steps was used to refine the criteria for clone classification. Phase II manipulated the subject system in order to explore the impact of different software system characteristics. Once again, feedback was used to refine the clone classification criteria. Phase III introduced a new judge in order to explore issues of judge variability and the impact of practice on the judge's decision-making. It also used a different subject system in order to further explore the effect of different subject systems. Phase IV re-ran the same trial as Phase III except with clone criteria which were less task-specific.

Results from a prior clone detection study conducted by Bellon [3] were used as a basis for this study. Bellon,

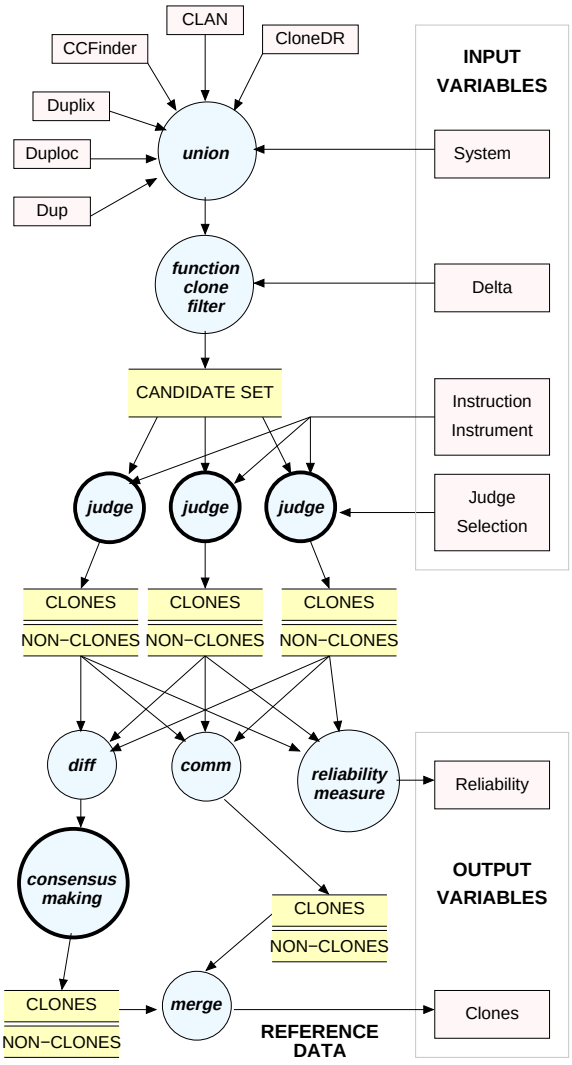


Figure 1. Basic flow of each study run

with the aid of 6 other groups of clone detector researchers worldwide, generated a clone detector result archive. The archive contains results from six detectors using a variety of detection techniques: Dup [1], CloneDR [2], CCFinder [7], Duplix [11], CLAN [14], and Duploc [5]. In addition, the Bellon study prepared 8 subject systems. Three of these systems are used in this study. They are:

SYSTEM	KLOC	DESCRIPTION
WELTAB II	11.5	Vote tabulation system
Cook v2.19	80.5	File construction (make-like)
SNNS v4.2	115.0	Neural net simulator & X11 GUI

All of these are written in `C`. The lines of code measure above (in thousands, i.e., KLOC) includes header lines, and the code is pre-processed to remove extra vertical whitespace and irrelevant or problematic preprocessor directives including `#define` and `#pragma` directives. One source line—a comment—within SNNS was modified to remove a character that broke our function parser.

5. Study phase I

Phase I of the study was intended to clarify a number of issues relating to the rest of the study. The primary question concerned how to construct suitable candidate clone sets for the judges to classify. In particular, we wished to determine a suitable value for δ. In addition, Phase I served a role much like a pilot study and initial exploratory study.We wished to ensure that we could come to consensus on how to classify the candidates. It was not obvious to us at the beginning that we would all agree on how to classify each clone. If we had troubles, we might have to switch to a different study design in Phase II. For instance, we might have had to implement a voting system rather than a consensus-generation step (as it turned out, a voting system was unnecessary, but we did not know this to begin with). We also wanted to use the initial runs to ensure the judges had a clear understanding of what should and should not be considered a function clone. If this were a psychology experiment, it might be called a practice session.

5.1. Data and subject systems

Bellon's processed WELTAB system was used as the subject system. Regarding candidate set selection, it would have been technically sufficient to start with all $n(n-1)/2$ combinations of function pairs, but clearly this is too conservative as even the relatively small Cook system would have nearly a million function pairs to sort through. Because the time of the judges is very valuable, a desirable quality of a candidate set is that it should be small, yet contain all potential clones or, failing that, it should be an unbiased sampling of the true clones. Prior studies, including those by Bellon [3] and Burd *et. al* [4] had taken the results from several clone detectors and merged them to start with a union set. The (implicit) hope is that using different detectors will alleviate the sampling bias that is almost sure to accompany any specific clone detector.

The results from Bellon's study included result sets from six different clone detectors. These clone candidates, however, are reported in terms of pairs of line segments from the program text. By inspection we knew the majority of these do not exactly match function bodies. Instead, most of them either identify some portion of a function body, or spill past the boundaries of function bodies. Some of the candidate clone pairs do not even reference code connected to a function body. Thus we needed an apparatus that could extract likely candidate function clones from line-oriented candidate data.

We decided to use a line coverage metric to determine membership in the candidate set. The basic idea is that each line-based clone candidate may be associated with some sub-range of two function bodies. The two function bodies are included in the candidate set if the average match between the two function bodies exceeds some threshold δ. Informally speaking, we are selecting function pairs if their coverage averages above some threshold. At $\delta = 1.0$ both functions have to be completely matched. At $\delta = .5$ only an average of half of the lines need to be.

More formally, let $\mathcal{C}_B$ be the list of candidate line-oriented clones generated by taking the union of all of the clone detector output from Bellon's study. Let $c \in \mathcal{C}_B$ be a line-oriented clone candidate that overlaps a pair of functions $< f_1, f_2 >, f_1 \neq f_2$. That is, if $\gamma(f_1, f_2, c)$ denotes the set of lines of function f_1 overlapped by c, then $\gamma(f_1, f_2, c) \neq \emptyset$ and $\gamma(f_2, f_1, c) \neq \emptyset$, else if c does not overlap both of f_1 and f_2 let $\gamma(f_1, f_2, c) = \emptyset$. Let $\Gamma(f_1, f_2)$ be the total coverage of function f_1 for function pair $< f_1, f_2 >$ given the candidate set $\mathcal{C}_B$, i.e., $\Gamma(f_1, f_2) = \bigcup_{c \in \mathcal{C}_B} \gamma(f_1, f_2, c)$. Then let $R(f_1, f_2) = |\Gamma(f_1, f_2)|/|f_1|$ be the ratio of lines of function f_1 covered by clones candidates, where $|x|$ denotes the size of x. Then the set of function clone candidates we wish to collect, $\mathcal{C}_\delta$ is given by the formula

$$\mathcal{C}_\delta = \{< f_1, f_2 > | \frac{R(f_1, f_2) + R(f_2, f_1)}{2} \geq \delta\}.$$

We did not know, however, what threshold to use. For instance, we worried that setting δ too high would artificially bias the selection against function clones with relatively sizable insertions or deletions. Conversely, a threshold too low would waste too much of the participants' time and thus reduce the scope of study possible given our restricted resources. Thus we first wanted data to use to calibrate δ for further studies. Using a simple clone candidate selection script, we generated three candidate sets to serve as study material. These were $C_1 = \mathcal{C}_{.95}$, $C_2 = \mathcal{C}_{.75} - \mathcal{C}_{.95}$, and $C_3 = \mathcal{C}_{.50} - C_2$. With these sets we hoped to see how the number of true clones dropped off as the allowable differences increased.

One of the main reasons we chose WELTAB as the first subject to study is that it contained a small enough number of candidates at $\delta = .50$ that we could examine them all. With the others, we needed to do sampling. Sampling in such cases was done by collecting a random sample of the functions from the subject system and restricting the full merged clone candidate set to only those clones related to the sampled functions. This method was chosen over the technique, exemplified by Bellon [3], of randomly sampling clone candidates from the full clone candidate set. Effectively our sampling technique generates a *full* clone candidate set from a random sub-set of the system.

5.2. Apparatus and materials

A simple program was implemented to present the clone candidates on the screen one at a time, and to record classification decisions. The program displayed the function clone candidates side by side while highlighting differences (using the GNU `sdiff` format). It allowed the user to go back and forth between candidates so that judgments could be revised. It recorded start and end times, logged all navigation and classification actions, and allowed stopping and restarting.

A small study booklet was employed which detailed the study procedure to follow, and provided guidelines and an example of how to classify candidate clones. Because the specific instructions to the judges may be critical for obtaining reliable and consistent results, and because the guidelines will be important in later phases, they are replicated in Figure 2.

5.3. Procedure

Judges were instructed and prepared in a group setting. They then completed the classification on their own time and without supervision. Afterwards, all of the judges were gathered together around a single computer and the consensus-building session began. One of us (Walenstein) acted as the experimenter during the instruction sessions and during the consensus-building session.

Individual sessions: instructions. Instruction and preparation consisted of verbal instructions describing the purpose of the study, the procedure to follow, the programs to run and how they are run, and the criteria to use to classify the candidates. The participants were instructed to run the candidate-presenting program for each of the three candidate sets. Only one of us (Walenstein) developed the session materials and detailed study design. Thus even though all of the judges were involved in the study and were aware of the research questions, an element of novelty was still present, so the instruction session was taken seriously. The judges were each given study booklets to refer to during the procedure. The participants were asked if they understood the procedure and were encouraged to ask any questions.

Individual sessions: protocol. The protocol for the individual study session required the judges to work in an environment with few distractions, and work for at most an hour at a time with a minimum 15 minute break between work sessions. Time, place, and pacing were uncontrolled and unsupervised.

Consensus-building session: instructions. At the beginning of the consensus-building session the experimenter explained that the goal of the session was to try to resolve all

FUNCTION CLONE: two functions are considered FUNCTION CLONES if and only if they CAN and PROBABLY SHOULD be refactored (i.e., reengineered) to either (a) eliminate one of them, or else (b) improve the source code for perfective engineering purposes.

An EXAMPLE of type (b) reengineering is when a function can be removed by abstracting the functions

```
int f1 ( int x ) { BLOCKA ; z = x + FL_UPD ; BLOCKB; }
int f2 ( int y ) { BLOCKA ; z = y + FL_DEL ; BLOCKB; }
```

(assume BLOCKA and BLOCKB are non-trivial blocks of code) might be reasonably reengineered to

```
int f12 ( int x, int inc ) { BLOCKA; z = x+inc ; BLOCKB; }
int f1 ( int x ) { return f12(x,FL_UPD); }
int f2 ( int y ) { return f12(y,FL_DEL); }
```

(or else assume the callee would pass in the right parameter).

Figure 2. Original function clone candidate classification definition

	#	Clones				% Clones				Non-Clones				% Non-Clones				Reversals				Total Time (min)			
Graph Label	**Cand.**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
WELTAB .95 (I)	156	154	146	141		99	94	90		2	10	12		1	6	8		0	18	2		27.7	37.5		
WELTAB .75 (I)	42	29	39	32		69	93	76		13	3	10		31	7	24		0	6	0		8.8	15.6	13.4	
WELTAB .50 (I)	53	10	14	11		19	26	21		43	39	42		81	74	79		0	1	0		7.6	6.7	13.7	
WELTAB .95 (II)	156	156	146	146		100	94	94		0	10	10		0	6	6		2	0	0		*937.2*	39.1	11.8	
WELTAB .75 (II)	42	40	34	31		95	81	74		2	8	11		5	19	26		0	2	2		10.1	*228.2*	11.5	
WELTAB .50 (II)	53	16	13	13		30	25	25		37	40	40		70	75	75		0	1	2		22.9	17.9	16.8	
COOK	316	196	6	151		62	2	48		120	310	165		38	98	52		121	6	1		2.0	48.5	56.7	
SNNS (III)	297	250	56	151	91	84	19	51	31	39	241	146	206	13	81	49	69	10	13	8	11	133.4	78.2	55.6	109.6
SNNS (IV)	176		41	47	36		23	27	20		135	129	140		77	73	80		8	0	0		38.8	*432.3*	71.4

Table 1. Raw data for individual classification sections, all phases

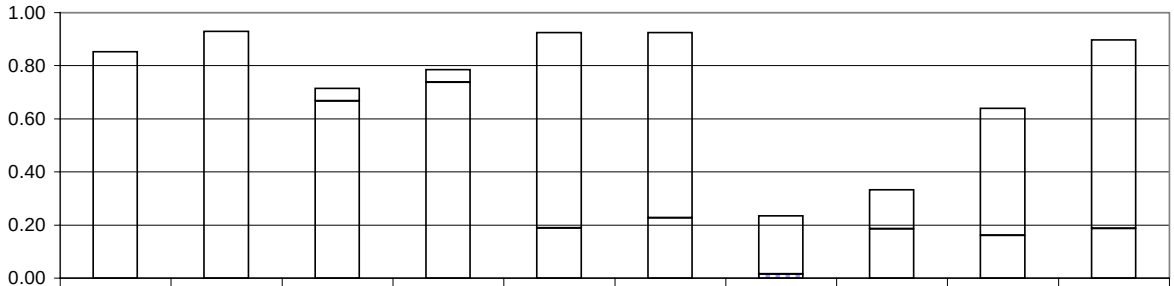
conflicting classifications. The participants were informed that they were expected to reach a consensus.

Consensus-building session: protocol. The protocol for the consensus-building session was effectively the same as for the individual sessions, except that instead of examining the full list of candidate clones, only the discrepancies were stepped through. Also, instead of a single judge, the judges as a group argued until they reached consensus. Time and place were mutually agreed upon in advance, and pacing was uncontrolled. On occasions when a participant appeared to disengage from the discussion, the experimenter firmly reminded each participant that they should not feel pressured to accepting an argument they did not believe, and gently encouraged them to keep arguing their position, if they felt like it. This was done in order to try to avoid having a dominant personality skew the results. Notes on decisions, rules, and guidelines were recorded at the end of the session (and during the session in Phase III and IV).

5.4. Results

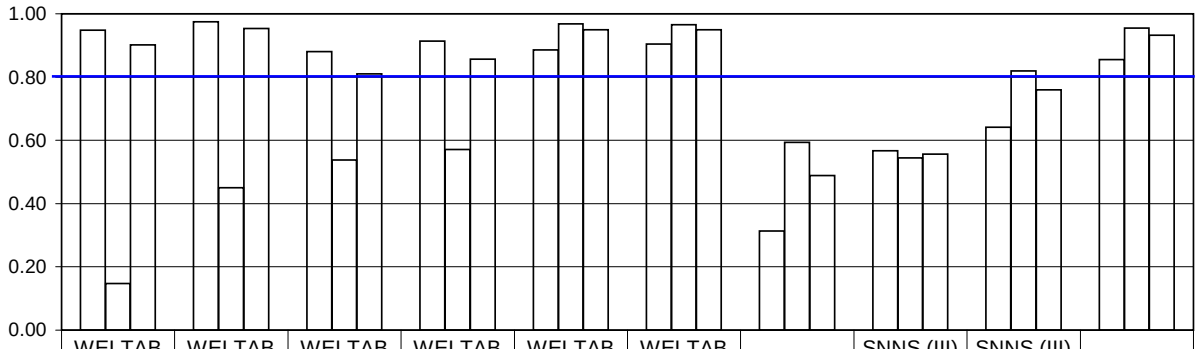
The raw data for classification activity in the individual trials appears in Table 1. The table lists the data for all phases: the top portion is for Phase I.[2] Numbers in italics indicate cases where the judge appeared to have accidentally left the experimental program running while not working on it. The column labeled "reversals" counts the number of times judges reviewed prior decisions, possibly to review or revise them. Figure 3 reports raw rater agreement values. Consider, for instance, the results for the C_2 candidate list (i.e., $\mathcal{C}_{.75} - \mathcal{C}_{.95}$) labeled WELTAB .75 (I). Out of 42 clone candidates (see Table 1), the three judges agreed initially that 67% of these candidates were clones and 5% of these were non-clones. The height of the bars in the diagram therefore is a rough indicator of agreement between judges on that particular data. Note that one may also read that the judges disagreed on $100 - 67 - 5 = 18\%$ of the candidates.

[2] N.B. Judge C classified only 155 of the 156 candidates for C_3.



	WELTAB .95 (I)	WELTAB .95 (II)	WELTAB .75 (I)	WELTAB .75 (II)	WELTAB .50 (I)	WELTAB .50 (II)	COOK	SNNS (III) [ABC]	SNNS (III) [BCD]	SNNS (IV)
☐ nonclones	0.00	0.00	0.05	0.05	0.74	0.70	0.22	0.15	0.48	0.71
☐ clones	0.85	0.93	0.67	0.74	0.19	0.23	0.02	0.19	0.16	0.19

Figure 3. Raw inter-rater agreement



	WELTAB .95 (I)	WELTAB .95 (II)	WELTAB .75 (I)	WELTAB .75 (II)	WELTAB .50 (I)	WELTAB .50 (II)	COOK	SNNS (III) [ABC]	SNNS (III) [BCD]	SNNS (IV)
☐ spec: clone	0.948	0.975	0.880	0.914	0.886	0.905	0.314	0.567	0.641	0.855
☐ spec: non-clone	0.148	0.450	0.538	0.571	0.968	0.966	0.593	0.544	0.820	0.955
☐ overall	0.902	0.953	0.810	0.857	0.950	0.950	0.489	0.556	0.760	0.932

Figure 4. Special and overall proportions of inter-rater agreement

Raw proportions of specific and overall agreement are reported in Figure 4. As is usual, specific overall agreement on clones (labeled spec:clone) can be interpreted as the conditional probability that the other judges will agree with one judge's opinion that some particular candidate is a clone. So, for example, if judge B classifies a candidate from the C_2 set as a clone, the other judges can each be expected to agree with an 88 in 100 chance. A horizontal line at the $.80$ value in Figure 4 indicates the commonly-used minimum threshold for inter-rater reliability.

5.5. Discussion

As Figure 3 indicates, all three judges agreed that as many as 1 in 5 clone candidates present in $\mathcal{C}_{.50}$ but not in $\mathcal{C}_{.75}$, were clones. In other words, even with a fairly sizeable number of insertions and deletions, some functions were considered clones of one another. Inspection of these candidate clones revealed matches of functions generating variations of what is logically the same textual report. The consensus reached by the judges was that these functions could naturally be merged into a single function which is parameterized by the report variant desired. Then the code variants would be selected by case analysis on a new parameter that indicated desired report type. Because of this information we felt it unwise to use a δ value any higher than $.50$ because we had evidence in hand that this sampling might under-represent clones with large insertions or deletions. Thus the candidate sets being sampled in later phases are all at the $\delta = .50$ level.

6. Study phase II

During the Phase I consensus-building process, the judges reviewed and discussed where their judgments differed. To reach consensus, therefore, they needed to make explicit the decision criteria we were using on these excep-

tion conditions. This led to the adoption of new classification criteria, which were added to the instructions. The instructions from Figure 2 were therefore augmented with the instructions from Figure 5.

6.1. Procedures, materials, subjects, and results

The procedures, apparatus, and data are the same as for Phase I, except that (1) a new description of the clone criteria was provided in the instruction booklet, and (2) four sessions were performed: the three WELTAB sessions from Phase I were repeated, and a session for the Cook subject system was completed (see below). Also, we did not go through the consensus-generating session for WELTAB sessions as in Phase I.

Two subject systems were used: WELTAB and Cook. Whereas the exact same full clone candidate set was used for WELTAB as it was for Phase I, a random sample of the candidate set for Cook was used. As for Phase I, individual session data, agreement measures, and proportions of agreement are included in Table 1 and Figures 3 and 4.

6.2. Discussion

The rater agreement measures increased for the re-run of the WELTAB session (several days separated the two rating sessions). The judges felt it was because the criteria for classifications were clarified, but it is impossible to be sure of the exact factors (it may be due completely to simple practice effects, for example).

The results from Cook were an eye-opener. We were frankly surprised at the level of disagreement. Out of 315 candidates, less than 1 in 4 were classified consistently by all three of us. Our overall agreement score of .489 is considerably lower than standard agreement guidelines of .800 or more. In fact, it was less than chance alone which, in hindsight, appears to mean that we were classifying on different criteria.

In the consensus-building session it quickly became clear why: we were disagreeing upon the purpose of the reference data set in relation to the code specifics. The Cook system is implemented in the C language, but it is designed and written as if it were C++. As a result, many of the clone were implementing constructors and destructors. Clearly, many of these were clones in the sense of being similar functions (e.g., sequences of initializations or memory freeing), but one of us (B) believed that it would be a mistake to reengineer these whereas others (A and to some extent C) did not consider that argument as relevant. B's argument was that it was important to preserve the independent maintainability of the various classes, and if one refactors the constructors and destructors it would create false couplings between unrelated classes, making independent maintenance implausible. This, of course, would be difficult for a simple clone detector to classify without knowing that the system was implemented to emulate C++ organization. Nonetheless, we decided to adopt B's reasoning because we wished to continue testing the viability of generating truly task-relevant reference data.

7. Study phase III

From Phase II we knew that many candidates were being rejected as false positives because they were not considered relevant for reengineering purposes. In addition, our collective experiences in arguing about the specific task-related criteria for classifying clones (several hours of discussion with many different examples to consider) left us skeptical that someone not privy to these consensus-building discussions would make similar agreements. This phase sought to explore these questions by adding a new judge who was unfamiliar with the prior classifications debates and examples.

7.1. Materials, procedures, and results

The sessions were as they were for Phase II, however a new judge was added. We used only the judges B, C and D for the consensus-building to in order to match the three-judge session characteristics from Phase II. As in prior phases, individual session data, agreement measures, and proportions of agreement are included in Table 1 and Figures 3 and 4.

7.2. Discussion

The observed reliability between judges improved with the addition of a new judge. From working through the examples and from looking through the raw judgment data from Table 1, it appeared to us that judge A was more incompatible with judges B and C than D was. This suggests that judge compatibility may be be a more serious factor than group judgment experiences or classification practice.

8. Study phase IV

As we worked through the consensus-building session in Phase III, we began to argue about—and then consider—the possibility of assuming a more task-independent set of criteria. After a lengthy discussion we decided to try changing the criteria such that candidates would be classified by *information in the source code alone*. So, for example, contextual information based on engineering context and programming style were not longer considered to be able to reject clone candidates as false positives. For instance, functions created by code generators might well be considered irrelevant for perfective maintenance purposes, yet

... when deciding on whether two functions are clones, consider the following issues, in order:

1. The refactoring must be worthwhile. If the clone pair is very simple or short it may not be worth the effort to reengineer them.
2. The functions should be able to be refactored in a simple and/or straightforward manner. Overly complicated parameterization is a sign that the candidates are unsuitable to be refactored.
3. The two functions should logically "the same" or "similar". For instance two functions might generate nearly identical error messages and only vary on a simple parameterizable condition, like one being fatal and calling an `exit()` function. It might make sense therefore to abstract the function into a more general error-message-generating function that can work in two modes: fatal and non-fatal.
4. Automatically generated code typically is not considered for refactoring.

These are guidelines, not rigid instructions. The experiment relies on you making your own judgment as to whether a candidate should be considered a clone or not. This decision may be affected by the particular software system, its coding styles, and application domain

Figure 5. Classification instructions added to the Phase I instructions for Phase II

were classified as clones in the reference set because the rejection criteria lay outside of direct source code analysis. The instructions to the judges were modified to reflect this new understanding. Specifically, we revised the phrase "PROBABLY SHOULD" in Figure 2 to read "POSSIBLY SHOULD", removed the criteria added in Figure 5, and added another instruction to classify candidates based solely on the source code.[3]

8.1. Materials, subjects, procedures and results

The new instruction booklets were distributed, and the individual sessions were re-ran for judges B, C, and D as in Phase III. Results are presented in Table 1 and Figures 3 and 4, as in previous phases. A different random sample of the SNNS clone candidate set was generated and used in this session.

8.2. Discussion

The inter-rater reliability jumped to higher levels similar to those we had seen in WELTAB. The belief of the judges was that the criteria for judging clones were easier to apply, i.e., there were fewer cases where the judge needed to be aware of—and take into account—issues outside of the source code proper. This, in part, relates to the fact that the judges did not need to make assumptions about reengineering contexts or the system's design and implementation. In short, the more restrictive, task-relevant guidelines were harder to apply. This fit in with our experiences in Phase II of the study, since many of these task-related issues did not surface until examining Cook.

[3]The workbook further instructed the judges to add sub-classifications in cases where task-relevant issues occurred, but we found the results uninformative and this aspect is omitted from discussion.

9. Conclusions and suggestions

This paper reports on a study on factors affecting the reliable generation of task-specific and task-independent function clone reference data sets. This is a relatively unexplored area of tool evaluation. The study is exploratory, and for that reason it has many weaknesses, including biased selection of participants, limited selection of subject systems, and uncontrolled experimental conditions. Such limitations obviously make firm generalization impossible. Yet it seems wise to take the difficulties we encountered as harbingers of genuine research problems in the area. We can also propose some suggestions for follow-up work.

Four research questions were investigated by our study (see Section 3). Our experiences and potential implications stemming from these questions are summarized below:

What clone classification criteria would be appropriate for benchmarking clone detectors? Some task-relevant reference data sets are likely to be easier to generate more reliably than some others. Although all of our reference sets were to some degree task-relevant (we excluded clones that were considered "too small" to be worth refactoring, for example), basing the classification criteria on concrete facts of the source code appears to help. We are at this point unsure, in fact, whether useful and reliable reference sets can be generated for many restricted task contexts. We note that our consensus-building sessions bore some resemblance to knowledge elicitation from experts: the discussion of disagreements helped elicit apparently implicit expert rules. This type of research might hold promise for knowledge-based clone detection.

A useful direction for future research is on understanding what kinds of reference data is difficult to generate, and what form of guidelines can be given to judges. At this point we can recommend (based on our experiences) that guidelines with several examples would be helpful, especially if

the examples illustrate instances where task-relevant issues might not be anticipated by the judges (e.g., the case of C++-like destructors written in C in Cook).

Is inter-rater reliability a problem? Inter-rater reliability is a potentially serious issue for many reengineering-related reference data sets. Given our experiences, a single judge cannot be trusted to give unbiased answers. This raises red flags about past reports of relevance and precision for clone detectors. It seems clear that inter-rater reliability measures should be calculated for human-generated reference data. In addition, after running this study, one suggestion we wish to make is that tool evaluations based on relevance and precision measurements might be modified to relate detector results to classification decisions from multiple judges. For instance, for SNNS we have the results of four different judges. Recall of clones for which all four judges agree might be weighted more than for candidates where only two can agree. A similar modification can be made for precision. Another way of viewing this suggestion is that instead of recall and precision, a clone detector might instead be evaluated according to its specific and overall reliability in reference to a collection of judges.

What is the role of practice effects? Our data does not support the idea that practice in classifying clones has a greater effect than the guidelines which are given to the judges. From our experience we can still recommend practice runs as a substitute for including many examples in the judges' instructions. We found it especially helpful to discuss disagreements. Thus one recommendation for tool evaluators is that, when generating reference sets, it might be helpful to have the experimenter pace the judges through their disagreements after a practice session.

Do system specifics make a difference? System specifics appear to have a serious impact on inter-judge reliability. The warnings relating to the second research question also apply to system-specific factors.

In closing, the study helped us understand some of the open questions regarding reference set generation and inter-rater reliability. Some of these results we found surprising. It would be helpful to employ follow-up studies using similar techniques. All experimental materials and results will be placed in a public and easily accessible benchmark and results archive [12].

References

[1] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1998 International Conference on Software Maintenance (CSM '98)*, pages 368–377, 1998.

[3] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Master's thesis, Fakultät Informatik, University of Stuttgart, Mar. 2002. Diplomarbeit Nr. 1998.

[4] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 36–43, 2002.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society Press, 1999.

[6] J.-F. Girard, R. Koschke, and G. Schied. A metric-based approach to detect abstract data types and state encapsulations. *Automated Software Engineering*, 6(4):357–386, 1999.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, July 2002.

[8] B. A. Kitchenham and S. L. Pfleeger *et. al*. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug. 2002.

[9] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 1997 Working Conference on Reverse Engineering*, pages 44–54. IEEE Computer Society Press, 1997.

[10] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00)*, pages 201–210, 2000.

[11] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'2001)*, pages 301–309, 2001.

[12] A. Lakhotia, J. Li, A. Walenstein, and Y. Yang. Towards a clone detection benchmark suite and results archive. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 285–286, 2003.

[13] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, Mar. 1997.

[14] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the IEEE Conference on Software Maintenance – 1996*, pages 244–254, 1996.

[15] B. S. Mitchell and S. Mancoridis. CRAFT: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the 2001 Working Conference on Reverse Engineering (WCRE'2001)*, pages 93–102, 2001.

[16] R. T. Rust and B. Cooil. Reliability measures for qualitative data: Theory and implications. *Journal of Marketing Research*, 31(1), Feb. 1994.