Evaluating Modern Clone Detection Tools

Jeffrey Svajlenko Chanchal K. Roy
Dept. of Computer Science, University of Saskatchewan, Canada
{jeff.svajlenko, chanchal.roy}@usask.ca

***Abstract*—Many clone detection tools and techniques have been introduced in the literature, and these tools have been used to manage clones and study their effects on software maintenance and evolution. However, the performance of these modern tools is not well known, especially recall. In this paper, we evaluate and compare the recall of eleven modern clone detection tools using four benchmark frameworks, including: (1) Bellon's Framework, (2) our modification to Bellon's Framework to improve the accuracy of its clone matching metrics, (3) Murakamki et al.'s extension of Bellon's Framework which adds type 3 gap awareness to the framework, and (4) our Mutation and Injection Framework. Bellon's Framework uses a curated corpus of manually validated clones detected by tools contemporary to 2002. In contrast, our Mutation and Injection Framework synthesizes a corpus of artificial clones using a cloning taxonomy produced in 2009. While still very popular in the clone community, there is some concern that Bellon's corpus may not be accurate for modern clone detection tools. We investigate the accuracy of the frameworks by (1) checking for anomalies in their results, (2) checking for agreement between the frameworks, and (3) checking for agreement with our expectations of these tools. Our expectations are researched and flexible. While expectations may contain inaccuracies, they are valuable for identifying possible inaccuracies in a benchmark. We find anomalies in the results of Bellon's Framework, and disagreement with both our expectations and the Mutation Framework. We conclude that Bellon's Framework may not be accurate for modern tools, and that an update of its corpus with clones detected by the modern tools is warranted. The results of the Mutation Framework agree with our expectations in most cases. We suggest that it is a good solution for evaluating modern tools.**

I. Introduction

Clone detection tools locate similar source code within a software system. By detecting and managing the clones in their software, developers can improve and maintain software quality, reduce development risks, prevent and detect bugs, and more [1]. The importance of clone detection is reflected by the number of tools published in the literature, and the frequency of new publications. A 2013 survey by Rattan et al. [2] found the existence of at least 70 tools, a 75% increase from the 40 tools found by Roy et al.'s [3] survey in 2009. Despite the large number of tools, there has been little evaluation and comparison of their performances.

Clone detection tools are commonly evaluated using the information retrieval metrics of recall and precision. While time consuming, precision is easily measured by validating a random sample of a tool's output. This is something that tool developers do, at least informally, while testing their tool. Recall has been much more difficult for developers to measure as it requires knowledge of the clones that exist in a software system. Reference data of true positive clones in a system may be built by (1) manually inspecting the system for clones, (2) manually validating the detection output of a variety of tools, or (3) inserting known clones into the system. Complete data is only possible by manual inspection, but this is non-trivial except for toy systems. Even a small system such as Cook, when considering only function clones, has almost a million function pairs to examine [4]. The other options are also difficult and time consuming, which is why tool developers usually do not report recall. While a small number of benchmarking frameworks have been published [5], [6], there has been a lack of recent tool evaluation studies that evaluate more than a few tools. Therefore, in this paper, **we evaluate the recall of eleven modern clone detection tools using Bellon's Framework and our Mutation and Injection Framework**.

Bellon's Framework [5] was created for an experiment that compared the performance of six tools contemporary to 2002. It uses a corpus of curated clones mined from 2% of the output of the participating tools. Murakami et al. [7] extended Bellon's Framework to improve the correctness of its type 3 recall measurement by making the framework gap aware. They manually identified the gap lines in Bellon's type 3 clone references, and modified the framework's clone matching metrics to ignore these gap lines. As part of this research, we propose a modification to Bellon's *ok* clone matching metric that improves its accuracy and corrects a fault.

Our Mutation and Injection Framework [6] evaluates clone detection tools using a corpus of artificially synthesized copy and paste clones. The clone generation process uses clone taxonomy based on a literature survey and validated by empirical evidence [3]. This ensures that the generated clones are both realistic and comprehensive. The framework is fully automated, so tools can be evaluated against a very large corpus without any manual effort required.

The contributions of this research are as follows. We execute and compare three versions of Bellon's framework, including: (1) Bellon's original version, (2) Murakami et al.'s [7] gap aware extension of the framework, and (3) our modification of the *ok* clone matching metric. We also evaluate the tools' recall using our Mutation Framework. Of concern is the accuracy of Bellon's Framework, as its reference data is based on clones detected by tools over a decade old (2002). These clones may not be compatible with modern clone detection preferences such as scope, granularity, or what constitutes a true positive clone. We evaluate our confidence in both benchmarks by (1) checking for anomalies in their results, (2) checking for agreement between the frameworks, and (3) checking for agreement with our expectations. We also compare our results with Bellon's Framework against those of Bellon et al.'s [5] experiment. Our expectations of the tools' recall are flexible and researched, including contact with some of the tool developers. While expectations may contain inaccuracies, they define our confidence in a benchmark's results. By comparing our expectations with two benchmarks, we can get a good idea of the tools' capabilities.

We found that the Mutation Framework measures high recall for many of the tools. Particularly, it suggests that ConQat, iClones, NiCad and SimCad are very good tools for detecting clones of all types. Many of the other tools also perform well. Clone detection users and researchers can consider these results, along with the features of the tools, to decide which tool is right for their use case. We find strong agreement between the Mutation Framework and our expectations, and suggest it is a good solution for measuring the recall of modern tools. Bellon's Framework frequently disagrees with our expectations and the Mutation Framework, often measuring considerably lower recall. We found anomalies in its results, including when we compare it against Bellon's original experiment. Our findings suggest that Bellon's Framework may not be accurate for modern tools, and that an updated corpus built by modern tools is warranted.

II. Definitions

Code Fragment: A contiguous segment of code from a software system. It is specified by a triple including the source file, start line and end line (inclusive).

Clone: A pair of code fragments that are considered similar by some definition of similarity.

Type 1: Code fragments that are syntactically identical, except for differences in white space, layout and comments.

Type 2: Code fragments that are syntactically identical, except for differences in identifier names, literal values, white space, layout and comments.

Type 3: Code fragments that are syntactically similar with differences at the statement level. The fragments may differ by the addition, removal or modification of statements.

Recall: The ratio of the clones within a software system that a tool is able to detect.

Precision: The ratio of the clones detected by a tool that are true clones and not false positives.

III. Bellon's Framework

Bellon's Framework is a product of Bellon et al.'s [5] clone benchmarking experiment, which measured the recall of six contemporary (2002) tools for four C and four Java systems. The framework uses a benchmarking corpus of real clones built by Bellon's manual verification ("oracling") of 2% of the 325,935 candidate clones detected by the tools. We use three variants of this framework including the original, Murakami et al.'s [7] gap-aware extension, and our version with an modified *ok* clone matching metric.

Bellon typified and added to the corpus only the true positives clones, as per his judgment, possibly with improvements to the clones' boundaries. This process was not formally specified, but from the experiment's publication [5], and from Baker's analysis of the experiment [8], we see that Bellon followed a number of rules: (1) minimum clone size of six lines including comments, (2) clone fragments may not start or end with comments, (3) clones must be replaceable by a function, (4) clones must be of the first three clone types, as defined in Section II, although Bellon additionally allowed type 2 clones to contain differences in expressions, (5) boundaries of accepted clones were expanded to the maximal size for their clone type, (6) clones capturing repetitive regions were left in the reporting style of the reporting tool. Due to disagreement over type 3 similarity requirements, no formal specification was used, and was instead left to Bellon's judgement.

Bellon's Framework automatically measures recall by mapping each of the clones detected by a tool (**candidates**) to one of the clones in the corpus (**references**). The mapping is produced using two clone matching metrics, the *ok* and *good* values, which measure how well two clones match with a value between 0.0 (total mismatch) and 1.0 (exact match). The framework maps each candidate to the reference that maximizes its *ok* and *good* values, with *good* taking precedence as the stricter metric. A candidate is considered an *ok* match of the reference it is mapped to if its *ok* value exceeds some given threshold p, similarly for *good* match. The framework reports *ok* recall and *good* recall as the ratio of the references that the tool captures by the *ok* and *good* matches, respectively.

The *ok* metric is shown in Eq. 1, and measures how well clone candidate C matches reference R. F_1 and F_2 are a clone's first and second code fragments, ordered by file name, start line, and then end line. The *ok* metric is based on the contain metric, Eq 2, which measures the ratio of F_A that is contained by F_B. For example, if both fragments are in the same file, and F_A includes lines 3 through 12 (inclusive) and F_B includes lines 7 through 14, then $contain = \frac{6}{10}$. The *ok* metric is the minimum containment of F_1 and F_2. It measures this for the optimal containment direction (C contains R, or R contains C) per fragment.

$$\begin{aligned} ok(C,R) = min(&max(contain(C.F_1, R.F_1),\\ &contain(R.F_1, C.F_1)),\\ &max(contain(C.F_2, R.F_2),\\ &contain(R.F_2, C.F_2))) \end{aligned} \tag{1}$$

$$contain(F_A, F_B) = \frac{|F_A \cap F_B|}{|F_A|} \tag{2}$$

The *good* metric is measured as in Eq. 3. It is based on the overlap metric, Eq 4, which measures the ratio of the unique source lines in F_A and F_B that are in both fragments. For example, if both fragments are in the same file, F_A includes lines 1 through 10 (inclusive), and F_B includes lines 5 through 15 (inclusive), then $overlap = \frac{6}{15}$. The *good* metric is the minimum overlap of the candidate's and reference's first and second fragments.

$$\begin{aligned} good(C,R) = min(&overlap(C.F_1, R.F_1),\\ &overlap(C.F_2, R.F_2)) \end{aligned} \tag{3}$$

$$overlap(F_A, F_B) = \frac{|F_A \cap F_B|}{|F_A \cup F_B|} \tag{4}$$

Gap Aware Version. Murakami et al. [7] suggest that type 3 recall can be measured more correctly by ignoring the gap lines in the type 3 references when evaluating the *ok* and *good* metrics. A tool is then evaluated for how well it matches only the cloned lines in a type 3 reference. To enable this, they manually inspected Bellon's type 3 references and identified their gap lines. The *ok* and *good* metrics are then modified to discard the reference's gap lines. Specifically, $C.F_1$ is replaced with $C.F_1 - G_1$, $R.F_2$ by $R.F_2 - G_2$, and similarly for $R.F_1$ and $C.F_2$ where G_1 and G_2 are the gap lines in the reference's first and second code fragments.

Our *Better-OK* Version. The *ok* match requires that either the candidate's or the reference's code fragments contain some

minimum ratio of the other, using the containment direction per fragment that maximizes this ratio. The critical flaw in the *ok* metric is that it accepts either containment direction. For benchmarking, we should only be interested in if the candidate contains some minimum ratio of a reference. Candidates that are contained by a reference may be a very poor detection of that reference. For example, consider a 30 line (per fragment) reference clone, and a 6 line candidate whose fragments are fully contained by the reference. This candidate has an *ok* metric of 1.0 for the reference, or a perfect *ok* match. The same is true even if the reference is 100 lines. Obviously this is a very poor match of the reference, and should not be accepted. We modify Bellon's *ok* metric to only consider the ratio of the reference contained by the candidate, as shown in Eq. 5. We call this the *better ok* metric or *b-ok* for short. To evaluate *b-ok* recall, we replace Bellon's *ok* metric, but do not modify the *good* metric or the clone mapping procedure.

$$b\text{-}ok(C,R) = min(contain(R.F_1, C.F_1)), contain(R.F_2, C.F_2))) \quad (5)$$

IV. Mutation & Injection Framework

Our Mutation and Injection Framework is an automated synthetic benchmark for evaluating clone detection tools. It generates a customizable corpus of artificial clones using a mutation analysis procedure. Mutation analysis is a well studied and accepted way of evaluating software tests, and we use similar methodologies to evaluate clone detection tools. Clone generation begins by extracting a code fragment from a repository of varied source code. The code fragment is duplicated and randomly modified by a mutation operator, which introduces a random edit following a clone taxonomy. The framework includes 15 operators spanning the 3 clone types, and is based on Roy and Cordy's [3] comprehensive and empirically validated taxonomy of the types of edits developers make on copy and pasted code. The original and mutated code fragments are injected into a unique copy of a subject system, evolving the system by a single copy, paste and modify clone. Per mutation analysis, this is repeated thousands of times to build a large corpus of mutant systems.

The subject tools are executed for these mutant systems, and their recall is measured specifically for the injected clones. Recall is judged by a flexible subsume-based clone matching algorithm that is parameterized by a subsume tolerance and a clone similarity threshold. The subsume tolerance considers a candidate to subsume a reference even if the candidate misses the first and/or last x% of the reference clone. The clone similarity threshold checks that the clone candidate is itself a true positive. Additional details about the framework, including a list of its mutation operators, can be found in our previous publications [6], [9].

V. Experiment

Bellon's Framework. We executed the tools for the framework's subject systems, and imported their results into the framework. We executed the framework's mapping and recall evaluation procedures using a clone matching threshold of 0.70. This is the value used in Bellon et al.'s [5] original experiment. The experiment was executed three times using Bellon's original clone matching metrics, Murakami's gap line metrics, and our 'b-ok' metric.

Mutation and Injection Framework. We evaluated the tools using two generated corpora, one Java and one C, of block granularity clones. For clone synthesis, we extracted code blocks from JDK6 and Apache Commons (Java), and the Linux Kernel (C). We injected the clones into IPScanner (Java) and Monit (C). For each corpus, we set the framework to randomly extract 250 code fragments, and mutate each using the 15 mutation operators, for a total of 3,750 clones. For each clone, 10 mutant systems were created using random injection locations, for a total of 37,500 unique mutant systems per corpus. We constrained the corpora to the following clone properties: (1) 15-200 lines in length per fragment, (2) 100-2000 tokens in length per fragment, (3) minimum 70% similarity measured by token and by line after type 1 and 2 normalization using a diff-based algorithm, and (4) mutations do not occur within the first and last 15% of a fragment (mutation containment). We selected the properties as the average default clone size and similarity defaults of the modern tools, which we believe estimates modern clone preferences. Typically, clone detection tools are slower for smaller minimum clone sizes. We needed to use a larger clone size than Bellon's to make execution of the tools for 37,500 systems practical.

For the tool evaluation, we used a subsume tolerance of 15%, and a minimum clone similarity of 60%. By setting the subsume tolerance to the same value as the mutation containment, we guarantee that any candidate clone accepted as a match of a reference has captured all clone type specific differences (mutations) in the reference clone. For comparison against the Bellon's Framework results, we summarized recall per language and per clone type by averaging the per mutation operator results. Due to limited space, we do not report recall per mutation operator in this paper. We do not execute Deckard for our Java corpus as it does not support the needed language specification (Java 1.6).

VI. The Participants

Eleven modern clone detection tools are investigated in this experiment. We used release date to judge the modernness of the tools. The oldest release of these tools was in 2006, while Bellon et al.'s [5] experiment was conducted in 2002. The participating tools are listed in Table I. Ideally, we would have included the tools of Bellon's original experiment as participants of our Mutation Framework benchmark to compare the results against Bellon et al.'s original experiment. However, Bellon's publications [5] [10] do not list the versions of the six participants, nor their configurations. Most of these tools are no longer available, or the available versions are now significantly updated (i.e., modern tools).

Configuration. Our goal is to benchmark the performance of these tools from a user perspective. We want the results to represent what an experienced user would receive for their own systems. An experienced user has explored a tool's documentation and is comfortable modifying the default settings as required for their use case. To emulate this user, we configured the tools by considering: (1) the tool's default settings, (2) the tool's documentation, and (3) the properties of the benchmark, including minimum clone size and clone types. For settings that are not well documented, we experimented with the tool to find an appropriate value. We avoided over-configuring or

TABLE I: Participating Tools

Tool	Language	†Expected Recall (Type) 1	2	3	Configuration for Bellon's Corpus	Configuration for Mutation Framework
CCFinderX 10.2.6.4 [11]	Java C	◕	◕	○	min. size: 25 tokens, min. token types: 6	min. size: 50 tokens, min. token types: 12
ConQat 2012.9 [12]	Java	◕	◕	◕	min. size: 6 lines, max. editing distance: 3, max. gap ratio: 0.30	min. size: 15 lines, max. editing distance: 3, max. gap ratio: 0.30
CPD 5.0.4 [13]	Java C	◕ ◕	◕ ○	○ ○	min. size: 30 tokens, literal/identifier normalization	min. size: 100 tokens, literal/identifier normalization
CtCompare 3.2 [14]	Java C	◕	◕	○	min. size: 30 tokens, max. isomorphic relations: 6	min. size: 100 tokens, max. isomorphic relations: 3
Deckard 1.2.3 [15]	Java C	◕	◕	◕	min. size: 30 tokens, 5 token stride, min. 90% similarity	min. size: 100 tokens, 4 token stride, min. 85% similarity
Duplo 0.2 [16]	Java, C	◑	○	○	min. size: 6 lines, min. characters/line: 1	min. size: 15 lines, min. characters/line:1
iClones 0.1.2 [17]	Java C	◕	◕	◕	min. size: 30, min. block size: 10, all transformations	min. size: 100, min. block size: 20, all transformations
NiCad 3.4 [18]	Java C	◕	◕	◕	clone size: 4-2500 lines, blind renaming, literal abstraction, function and block clones, max. 30% dissimilarity	clone size: 10-2500 lines, blind renaming, literal abstraction, function/block clones, max. 30% dissimilarity
Scorpio 2011 [19]	Java	◕	◕	◕	min. size: 6 statements, normalize identifier/literal to type	min. size: 15 statements, normalize identifier/literal to type
SimCad 2.2 [20]	Java C	◕	◕	◕	consistent identifier renaming, function/block clones	consistent identifier renaming, block clones
Simian 2.3.34 [21]	Java C	◕	◕	○	min. size: 6 lines, normalize literals/identifiers	min. size: 15 lines, normalize identifiers/literals

†e.g., ◕◑◔ = 75% Type 1 Recall, 50% Type 2 Recall, 25% Type 3 Recall

over-optimizing the tools for the benchmark, as a user would not be able to do this for their own software systems. We also avoided configuring the tools in a way that would maximize recall at the sacrifice of precision. Generally, we configured the tools for a benchmarks' minimum clone size, and enabled type 2 normalization features. The tool configurations for each benchmark are summarized in Table I. Different configurations are required due to differences in the benchmarks' minimum clone size. Since the properties of Bellon's corpus are not well known, we used more permissive settings with it.

Different configurations may result in better or worse recall for these tools. Wang et al. [22] refer to this as the confounding configuration choice problem. They propose the use of a genetic algorithm for finding tool configurations that optimize the tools' agreement on what is and isn't clone code in a software system. Using Bellon's Framework, they demonstrate that their configurations have a higher recall than the tools' default configurations. In general, compared to the default settings, their algorithm reduced minimum clone size and enabled type 2 normalization features. Our strategy altered the default configurations in a similar way, although our settings are a little more cautious to prevent loss of tool precision. We also found that that our targeted configurations perform better then the tools' default settings with Bellon's Benchmark. Some of the tools' default configurations were optimized for demonstration use (short execution time). Our configurations may be more appropriate for benchmarking as configurations that optimize agreement between the tools may restrict the individual tools' unique detection characteristics and strengths.

Recall Expectations. Before we executed the benchmarks, we evaluated our expectations of each tool's recall, which are summarized in Table I. We assigned expected recall in 25% increments, starting at 0% but capped at 90%. We consider a measured recall to agree with our expectation if it is within $\pm$12.5% of the expected value. This strategy gives our expectations flexibility, as our expectations are educated estimates. If agreement is found, then we are confident that the expectation and benchmark are correct. Otherwise, we suspect that either our expectation and/or the benchmark is inaccurate.

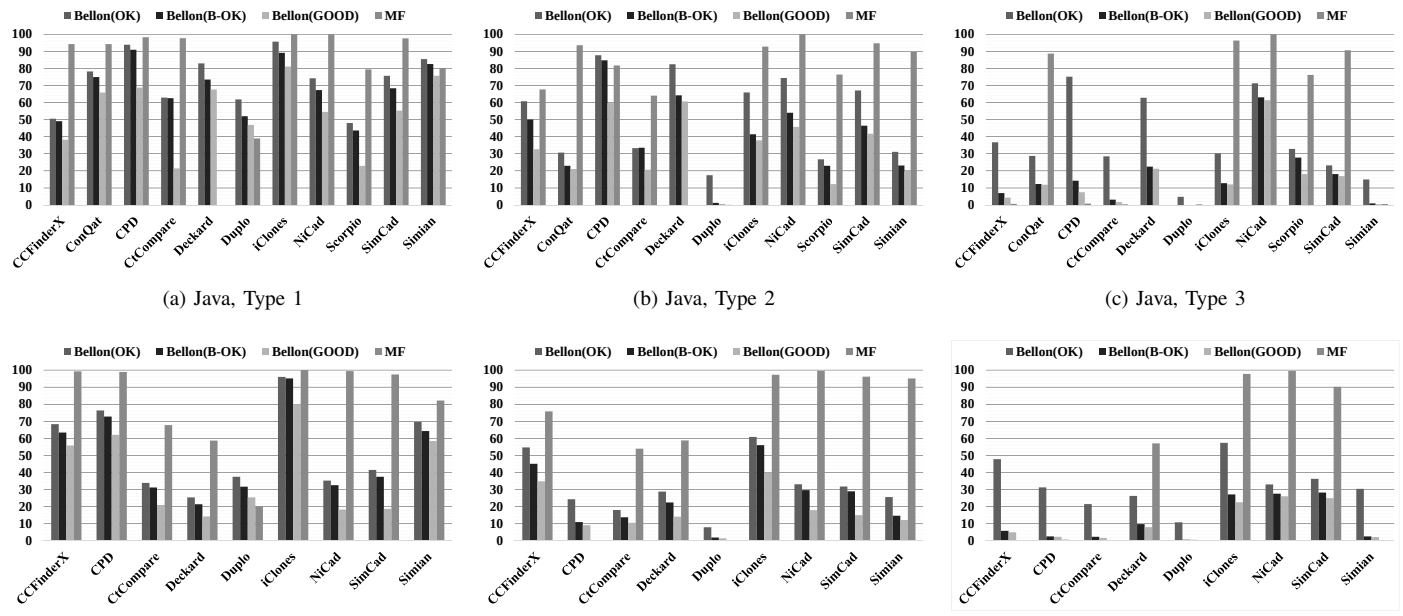
We chose our expectations by consulting the tools' documentation, publication, and literature discussion [2], [3]. For type 1 and 2 recall, we considered the normalization features directly or indirectly supported by the tools. For type 3 recall, we considered the tools' similarity metrics and recommended sensitivity. We also considered our experiences with these tools in our other studies. Where possible, we reached out to the tool developers for their opinions of our expectations. We were optimistic about the quality of the tools, and of the frameworks' ability to evaluate them. Despite these efforts, the expectations may still contain inaccuracies or be controversial between clone researchers. This is why we use a generous window (25%) around the expectation when determining agreement. These expectations give us the ability to uniformly evaluate our confidence in the benchmark results.

VII. Results

We present and discuss the performance of the tools as measured by Bellon's Framework in Section VII-A. We comment on observed differences between the *ok* and *good* metrics, and discuss some anomalies in the results. In Section VII-B we discuss how type 3 recall changes when measured by Murakami et al.'s [7] gap aware extension of the framework, and the performance of the tools using our *b-ok* metric in Section VII-C. We compare our results with the modern tools against Bellon et al.'s [5] original experiment in Section VII-D. We then compare our results with the variants of Bellon's Framework against our expectations for these tools in Section VII-E. In Section VII-F we present and discuss the recall of the tools as measured by the Mutation Framework, and compare them against our expectations. We compare the results of the two frameworks in Section VII-G. The recall measurements of the two frameworks are summarized in Figure 1, and compared against our expectations in Table II. We summarize agreement between the results and our expectations in Table VI, and agreement between the frameworks in Table VII.

A. Bellon's Framework Results - Original Framework

Java Type 1. Using the *ok* metric, most of the tools have a recall exceeding 70%, with CPD and iClones exceeding 90%. Scorpio performs poorly, detecting less than 50%, while CCFinderX only barely exceeds 50% recall. CtCompare and Duplo obtain fair results, with a little more than 60% recall. Many of these tools' recall drops considerably when the *good*



(a) Java, Type 1 (b) Java, Type 2 (c) Java, Type 3

(d) C, Type 1 (e) C, Type 2 (f) C, Type 3

Fig. 1: Measured Recall - Benchmark Results

TABLE II: Expected Vs. Measured Recall: Mutation Framework (MF) and Bellon's Framework (*ok*, *b-ok*, *good* metrics)

	Tool	CCFX			ConQat			CPD			CtComp.			Deckard			Duplo			iClones			NiCad			Scorpio			SimCad			Simian		
Language	**Clone Types**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
Java	Expected	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○
	MF	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	—			◔	○	○	●	◕	◕	●	●	●	◕	◕	◕	◕	◕	◕	◕	◕	○
	ok	◑	◑	◔	◕	◔	◔	◕	◕	◕	◑	◔	◔	◕	◕	◑	◑	◔	◔	◕	◑	◔	◕	◕	◕	◑	◔	◔	◕	◑	◔	◕	◔	◔
	b-ok	◑	◑	◔	◕	◔	◔	◕	◕	◔	◑	◔	○	◕	◑	◔	◑	○	○	◕	◔	◔	◕	◑	◑	◑	◔	◔	◑	◔	◑	◔	◕	◔	○
	good	◔	◔	◔	◕	◔	◔	◑	◑	◔	◔	◔	○	◕	◑	◔	◑	○	○	◕	◔	◔	◑	◑	◑	◔	◔	◔	◑	◑	◔	◕	◔	○
C	Expected	◕	◕	○	—			◕	○	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	—			◕	◕	◕	◕	◕	○
	MF	●	◕	○	—			◕	○	○	◑	◑	○	◑	◑	◑	◔	○	○	●	◕	◕	●	●	●	—			◕	◕	◕	◕	◕	○
	ok	◑	◑	◑	—			◑	◔	◔	◔	◔	◔	◔	◔	◔	◔	◔	◔	●	◑	◑	◔	◔	◔	—			◑	◔	◔	◑	◔	◔
	b-ok	◑	◑	◔	—			◑	◔	◔	◔	◔	○	◔	◔	◔	◔	○	○	●	◑	◔	◔	◔	◔	—			◑	◔	◔	◑	◔	○
	good	◑	◔	◔	—			◑	◔	◔	◔	◔	○	◔	◔	◔	◔	○	○	◕	◑	◔	◔	◔	◔	—			◔	◔	◔	◑	◔	○

◕ = Indicates recall (0-100%) as ratio of pie filled (e.g., in this case, 75%).

metric is used. Only iClones and Simian exceed 70% recall with the good metric.

C Type 1. Most of the tools have poorer type 1 detection for C than Java. The exceptions are CCFinderX, which performs better for C, and iClones, which has comparable results for both languages. Only CPD, iClones and Simian exceed 70% recall with the *ok* metric. CCFinderX is just shy of 70% with the *ok* metric, while the remainder fall below 50%. Only iClones manages a recall over 70% with the *good* metric, while CCFinderX, CPD and Simian have a recall greater than 50%.

Java Type 2. Only CPD, Deckard and NiCad exceed 70% recall by the *ok* metric. CCFinderX, iClones and SimCad exceed 50%, while the remainder fall below 35%. For the *good* metric, CPD and Deckard manage to maintain a recall around 60%, while the remainder fall below 50%.

C Type 2. Again, the tools generally perform worse for C clones. For the *ok* metric, only iClones and CCFinderX exceed 50% recall, while all the tools fall below 50% when the *good* metric is used.

Java Type 3. For the *ok* metric, only CPD and NiCad achieve a recall greater than 70%, with CPD just reaching 75%. Of the remaining tools, only Deckard exceeds 50%. With the *good* metric, NiCad achieves 60%, while the remainder fall below 25%. A notable anomaly is the type 2 detectors' sizable type 3 recall with the *ok* metric, especially CPD (75%). This is resolved by the good metric.

C Type 3. Only iClones exceeds 50% *ok* recall, with CCFinderX nearly meeting 50%. None of the tools have a *good* recall more than a little beyond 25%. Some of the type 2 detectors are achieving a type 3 *ok* recall, while their type 3 *good* recall is correctly near 0%. Type 3 detectors Deckard and NiCad have much better type 3 performance for the Java clones, while iClones and SimCad perform better for C.

In most cases, recall measured using the *good* metric is considerably lower than recall measured by the *ok* metric. Bellon's Framework suggests that the modern tools are bad

at detecting a clone's precise boundaries. However, for the type 1 and 2 clones, this may be due to Bellon's oracling process. When he accepted a clone he would change the clone's boundaries (if needed) to the maximal size of its reported clone type [8]. It is possible that the accepted clone could have been contained within a larger clone of a higher clone type. Likely, the tools prefer to report the larger clone of the higher type. The *ok* metric (contain) will accept this larger clone as a match, but the *good* metric (precise capture) is likely to reject it. For this reason, the *good* metric may be too strict for the type 1 and type 2 clones. However, our proposal of the *b-ok* metric shows why the *ok* metric is too permissive.

However, the *ok* metric may be too weak for measuring type 3 recall. A number of type 2 detection tools were receiving sizeable type 3 recalls with the *ok* metric. Since these tools cannot detect type 3 clones, the *ok* metric must be permissive enough to sometimes accept clones which only capture the type 1 or 2 portions of the type 3 clone. When measuring type-specific recall, this is undesirable. The *good* metric was appropriately measuring near-zero recall for these tools.

Overall, we found that many of the tools performed well for type 1 Java clones, and a few had good performance for type 2 Java clones, when the *ok* metric is used. Performance was generally weaker for C clones, and most tools performed poorly for both language's type 3 clones. The type 2 clone detectors were surprisingly able to achieve a type 3 recall when the *ok* metric was used, while the type 3 detectors are struggling to detect type 3 clones, even when measured by the permissive *ok* metric.

B. Bellon's Framework Results - Murakami Extension

Murakami et al. [7] suggest that type 3 recall is more correctly measured when Bellon's *ok* and *good* metrics ignore the gap lines in the type 3 references. We found that ignoring the gap lines has minimal impact on the tools' *ok* and *good* type 3 recall. Compared to Bellon's original metrics, the tools' gap-ignoring type 3 recall has an absolute change of no more than $\pm 1.5\%$, with two exceptions. CPD's type 3 *ok* recall for Java has an absolute increase of 7.2%, and iClones's type 3 *good* recall for C an increase of 3.7%. Ignoring these outliers, the average absolute change was $\pm 0.48\%$. This means that it is extremely rare for these tools to fail to capture a type 3 reference due to reporting only the cloned regions but not the non-cloned regions.

Murakami et al. investigated this difference for NiCad, Scorpio and CDSW. Their experiment found significant differences in type 3 recall for Scorpio and CDSW. Their experiment agrees with ours that ignoring gap lines has negligible effect on NiCad's recall. Our disagreement over Scorpio may be due to how we handled Scorpio's output. Of our subject tools, Scorpio is unique in that it does not report code fragments as source line regions. It is PDG-based, and reports code fragments as sets of (possibly non-sequential) program elements. We converted these to continuous line regions using the source lines of the earliest and latest program elements as the start and end lines. Murakami et al. do not mention how they handled this in their experiment.

While ignoring gap lines had minimal effects in our experiment, Murkami's gap line data is still valuable. It can be used to evaluate the correctness of clone detection tools that identify gap lines in their reported clones. Bellon's Framework is designed to handle clone detection tools that report code fragments as continuous source line regions. Knowing the locations of gaps in the type 3 references may make it possible to adapt Bellon's Framework to support tools that report code fragments as discontinuous source line regions without (approximate) conversions of their output.

TABLE III: OK to BetterOK - Relative Change in Recall

		Java Clone Types (%)			C Clone Types (%)		
Tool		**1**	**2**	**3**	**1**	**2**	**3**
1	Duplo	-16	-93	-100	-15	-75	-94
	CPD	-	-	-	-5	-55	-92
2	CCFinderX	-3	-18	-81	-7	-18	-88
	CPD	-3	-3	-81	-	-	-
	CtCompare	-1	1	-89	-8	-24	-89
	Simian	-3	-26	-94	-8	-43	-91
3	ConQat	-4	-25	-57	-	-	-
	Deckard	-11	-22	-64	-16	-22	-63
	iClones	-7	-37	-58	-1	-8	-53
	NiCad	-9	-27	-12	-8	-10	-16
	Scorpio	-9	-14	-15	-	-	-
	SimCad	-10	-31	-22	-10	-9	-22
Avg: Tool Doesn't Support		-	-93	-89	-	-65	-91
Avg: Tool Does Support		-7	-21	-38	-9	-19	-39
Avg: All		-7	-27	-61	-9	-29	-68

C. Bellon's Framework Results - The Better OK Metric

The recall of the tools using our *b-ok* metric are compared against Bellon's *ok* recall in Figure 1. The relative change in recall going from Bellon's *ok* recall to our *b-ok* recall is summarized in Table III, with the tools grouped by the maximum clone type they are able to detect. We also average the relative change across the tools, including specifically for tools that support or do not support particular clone types. The tools' recall decreases when our *b-ok* metric is used, with the exception of a marginal increase in CtCompare's Java type 2 recall. The decrease in recall means that there are candidates reported by the tool that are 70% contained by references in the benchmark, but none of these candidates contain 70% of these references. Likely, the tool reported only a small ($<$70%) portion of these references, which is why our *b-ok* metric rejects them as a match. While recall generally decreased, an increase is possible (CtCompare) because Bellon's Framework maps each candidate to the reference that maximizes its *good* and then *ok* values. By replacing the *ok* metric by the *b-ok* metric, the mapping can change in such a way that more references are matched given the matching threshold (70%).

Tools lacking type 3 support lose the majority of their recall (81-100%), and similarly for tools lacking type 2 support (55-93%). This improves the anomaly we found in Bellon's *ok* recall for which the tools' have sizable recalls for clone types they don't support. With Bellon's *ok* metric, a tool could match a type 3 reference by reporting a candidate that captures even a minuscule type 1 or 2 region within the reference, or a minuscule type 1 region in a type 2 reference. Our improved metric will only accept these cases when the detected lower clone type region is at least 70% of the reference clone. This appears to be rare, as the average *b-ok* recall for clone types a tool does not support is 4% with a maximum of 15%.

On average, the relative decrease in recall is larger for

TABLE IV: New Vs. Old Experiment - OK Metric

Java		OK Metric 1	OK Metric 2	OK Metric 3	GOOD Metric 1	GOOD Metric 2	GOOD Metric 3
AVG	**new**	73.6	52.5	37.2	54.4	32.1	14.1
	old	61.6	48.9	26.5	43.5	33.0	4.1
MAX	**new**	95.6	87.7	75.1	81.1	60.7	61.4
	old	94.9	85.3	61.6	67.3	48.9	7.5
C		**1**	**2**	**3**	**1**	**2**	**3**
AVG	**new**	53.8	31.7	32.8	39.3	17.3	10.4
	old	52.8	36.6	32.9	41.7	25.8	8.8
MAX	**new**	96.0	60.9	57.4	79.9	39.9	26.1
	old	85.7	79.8	68.3	79.0	68.1	22.2

higher clone types, considering only the tools that support the respective types. Type 3 recall reduction was significant for some of the type 3 tools, with ConQat, Deckard and iClones losing over half of their recall when our *b-ok* metric is used. Even these advanced tools are only reporting small (<70%) regions of the references that were matched by Bellon's *ok* match. What is unknown is if this is due to a deficiency in these tools, or disagreement with Bellon's definition of a type 3 clone, particularly the amount of dissimilarity allowed. The other type 3 tools had less significant reductions (12-16% relative decreases). Overall, our correction to Bellon's *ok* metric has a considerable effect on the measured recall.

D. Bellon's Framework - Modern Vs. Original Experiment

Our expectation is that clone detection has significantly evolved since Bellon's original experiment in 2002, especially for type 3 detection. In this section, we examine if Bellon's Framework supports this expectation by comparing our results with the modern tools against the results of Bellon's original experiment. In order to compare the two experiments, we calculated the average and maximum *ok* and *good* recalls across the tools of the individual experiments (Table IV).

Java. In general, the new tools outperform the old for the Java clones with both the *ok* and *good* metric. However, the average recall of the new tools is not as significant of an improvement as we expected. At most, the modern tools lead the old tools by 12%, and fall behind by 1% in one case. Considering the best performing of the new and old tools (maximum), the new tools perform consistently better. The increase in maximum recall is marginal with the *ok* metric, except for type 3 clones (+15%). The new tools have a considerably higher maximum *good* type 3 recall (+54%). This advantage is from NiCad, while the other modern tools had negligible *good* type 3 recall.

C. For the C clones, both the new and old tools have a very similar average recall by both metrics, with the old tools having up to 10% better recall for type 2 clones. Considering the best performing of the tools, the old tools mostly perform better, with the new tools having only a slight advantage in type 3 detection with the *good* metric (+3.9%).

The difference in type 3 recall between the old and new tools is strange. Only for the Java clones by the *good* metric does the best of the new tools outclass the old tools for type 3 detection. In the other cases, the difference is only 3-15%, with the best of the newer tools performing worse for type 3 C clones by the *ok* metric. Type 3 clone detection has been an area of focus in clone detection since Bellon's original experiment, so we expected the new tools to perform much better than the old. This suggests that Bellon's corpus does not have sufficient type 3 representation to accurately judge these modern tools.

TABLE V: CCFinder vs. CCFinderX

Clone Types	OK 1	OK 2	OK 3	GOOD 1	GOOD 2	GOOD 3
CCX-Java	50.5	60.7	36.7	38.2	32.7	4.3
CC-Java	88.0	85.3	30.9	42.5	48.9	6.3
CCX-C	68.3	54.7	47.8	55.8	34.9	5.0
CC-C	85.7	79.8	68.3	79.0	68.1	13.0

These two experiments are not exactly equivalent. The old tools have the advantage that Bellon's clone references are based off the clones the old tools detected. However, the modern tools have the advantage of up to a decade of clone detection research. Even if the the new tools did not contribute to the benchmark, they are the state of the art and should not have a problem detecting clones found by their predecessors.

It is interesting to compare CCFinderX to its direct predecessor, CCFinder, that participated in the original experiment (Table V). We can reasonably assume that CCFinderX (2009) should be an improvement over CCFinder (2002). However, CCFinderX's recall is considerably lower than CCFinder's for all clone types and both metrics, with the exception of a 6% lead in Java type 3 *ok* recall. The exception is likely an anomaly, as both versions of CCFinder lack type 3 support. In the original experiment, CCFinder was executed for its default settings, while we executed CCFinderX with more permissive settings than its modern default. It is possible that CCFinderX's core algorithm is less aggressive in detecting clones, possibly to increase precision. Or perhaps CCFinderX's preferences of what constitutes a true positive clone has changed, and disagrees with Bellon's definitions. Having a previous version that contributed to the corpus, we expected CCFinderX to be somewhat attuned to the benchmark. That CCFinderX performs considerably worse than CCFinder suggests that clone preferences have changed, and that the modern tools cannot be accurately judged by Bellon's corpus.

E. Bellon's Framework Variants Vs. Expectations

In this section we compare the tools' recall as measured by Bellon's Framework, using both its original and our improved metrics, against our expectations for these tools. Since we created our expectations in 25% increments, we consider measured recall to agree with our expectations if their absolute difference is 12.5% or less. Measured recall is compared against our expectations in Table II, and their agreement is summarized in Table VI.

Type 1. The *ok* recall agrees with our expectations for 6 of the 11 Java tools, but for only 2 of the 9 C tools. Three of these tools lose agreement when our *b-ok* metric is used. In the cases of disagreement, the recall measurements are generally considerably lower than our expectations. Only iClones (Java and C) and Duplo (Java) agree with our expectations with the *good* metric. The *good* recall of the remainder is considerably lower than our expectations. We expected these tools (with the exception of Duplo) to have a type 1 recall around 90%. These tools remove type 1 differences when they parse or pre-process

TABLE VI: Agreement Between Measured and Expected Recall, Mutation Framework (MF) and Bellon's Framework

Language	Tool	CCFX			ConQat			CPD			CtComp.			Deckard			Duplo			iClones			NiCad			Scorpio			SimCad			Simian			% Agree
	Clone Types	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	
Java	Expected	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	-
	MF	⊗	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	○	⊗	⊗	⊗	⊗	⊗	⊗	90.0%
	ok	○	○	○	⊗	○	○	⊗	⊗	○	○	○	○	⊗	⊗	⊗	⊗	○	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	⊗	○	○	30.3%
	b-ok	○	○	⊗	○	○	○	⊗	⊗	○	○	○	⊗	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	⊗	○	⊗	30.3%
	good	○	○	⊗	○	○	○	○	○	⊗	○	○	⊗	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	○	○	⊗	24.2%
C	Expected	◕	◕	○	—			◕	○	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	—			◕	◕	◕	◕	◕	○	-
	MF	⊗	○	⊗	—			⊗	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	74.1%
	ok	○	○	○	—			○	○	○	○	○	○	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	○	14.8%
	b-ok	○	○	⊗	—			○	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	⊗	29.6%
	good	○	○	⊗	—			○	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	⊗	29.6%

⊗ = Agree (±12.5%) ○ = Disagree ○◔◑◕● = Recall (0, 25, 50, 75, 90%)

input code. By configuring the tools with the benchmark's minimum clone size, it should be trivial for these tools to detect the simple type 1 clones. At the very least, the tools should detect the type 1 references as components of larger type 2 or type 3 clones, which would be accepted by the *ok* and *b-ok* metrics if not the *good* metric. It is strange that recall is further from our expectations in the C cases, despite the tools advertising full C language support.

Type 2. Recall measurements agree with our expectation for tools that do not support type 2 clones (Duplo, CPD for C), at least by the *b-ok* and *good* metrics. Of the tools supporting type 2 detection, CPD's (Java) agrees with our expectations by both the *ok* and *b-ok* metrics, as well as Deckard (Java) by only the *ok* metric. Otherwise, none of the tools' type 2 recalls agree with our expectations. Generally, their type 2 recall is considerably lower than our expectations. We expected many of these tools to have near 90% recall for type 2 clones. Most support the required type 2 normalizations (literal values, identifier names), which reduces type 2 detection to simple type 1 detection. It is therefore suspicious that not only do these results not agree with our expectations, but that the results do not mirror the type 1 results. This is at least partially due to Bellon's oracling process, which allowed type 2 clones to have an identifier or literal in one fragment be replaced by an expression in the other, as found by Baker [8]. The modern tools would consider these replacements to be near-miss gaps, and consider them to be type 3 clones. In this case the type 2 detectors would fail to report them, while the type 3 detectors may find them similar enough to report. However, this oracling error may not be too pronounced as the best performing Java type 2 detector is CPD, which lacks type 3 support.

Type 3. Considering the type 2 detectors, the *b-ok* and *good* recalls agree with our expectations (0% recall), with the exception of CPD for Java clones and the *b-ok* recall. The strong agreement is because these metrics generally will not accept a type 2 candidate as a match of a type 3 reference. With the exception of duplo, the type 2 detectors' type 3 *ok* recall does not agree with the expectation. As mentioned previously, the *ok* metric is allowing candidates detecting even only small regions of the type 3 clone as matches. Considering the type 3 detectors, none of the tools' type 3 recalls agree with our expectations. The only exception is Deckard whose Java type 3 *ok* recall agrees with our expectations. However, its *b-ok* and *good* Java type 3 recalls are considerably lower than our expectations. Considering Bellon's type 3 references were found by tools contemporary to 2002, and modern tools are considered to excel at near-miss clone detection, it is strange that they perform so far under our expectations. Perhaps the corpus simply does not have a large or diverse enough representation of type 3 clones to evaluate modern tools. Or perhaps Bellon's type 3 references disagree with the modern tool's type 3 clone preferences (e.g., scope, minimum similarity, true vs. false positive, etc.).

Recall measured by Bellon's Framework is generally lower than our expectations. A common belief in the clone community is that our clone detection techniques are very mature and have high recall. The disagreement between our expectations has two possible and potentially overlapping conclusions: (1) the modern clone detection tools are not as proficient as we believe, i.e., our expectations are incorrect, or (2) Bellon's Framework does not accurately measure the performance of modern tools. To gain further insight into this question, we consider the results of the Mutation Framework below.

F. Mutation Framework Results vs. Expectations

In this section we discuss the recall of the tools as measured by the Mutation Framework, and compare these against our expectations for the tools. As the granularity of our expectations was 25%, we consider the measured and expected recalls to agree if they have an absolute difference no greater than 12.5%. The tools' recall by the Mutation Framework are shown in Figure 1, and compared against our expectations in Table II. Agreement with expectations is summarized in Table VI.

Type 1. The Mutation Framework measures a very high recall ($>$ 90%) for most of these tools across both languages. Scorpio's and Simian's recall is a little lower at 80%. These results agree with our expectations of the tools. Duplo has poor type 1 recall, as it doesn't normalize for formatting differences, which is within our expectations for Java but not for C. While CtCompare has strong recall for the Java clones, its type 1 performance was considerably weaker for the C clones, and outside of our expectations. Deckard's recall for C is also below our expectations, at 59%.

Type 2. The framework also measures very high recall ($>$90%) for most of the tools that support type 2 detection. CPD falls a little behind the top performers, with a Java recall of 81%. These results match our expectations. The framework correctly identifies that Duplo and CPD (for C) do not support type 2 clones, with a near 0% recall. CtCompare does not

support literal value normalization and recommends limits on identifier normalization, so we are not surprised by its lower recall for Java (64%), although its recall for C (53%) is lower than anticipated. Scorpio's recall falls just outside our expectations, with a recall of 76%. CCFinderX is also less than our expectations, with 67% for Java and 76% for C. Deckard's type 2 recall matches its type 1, 59% and is considerably less than we expected.

Type 3. iClones and NiCad have near-perfect recall (>95%) for both languages, while ConQat (89%) and SimCad (90%) also achieve very high recall. These results match our expectations for these tools. The framework correctly identifies the tools that lack type 3 support, with near 0% recall. The framework measures recall outside of our expectations for Scorpio (76%) and Deckard (56%).

Compared to the other type 3 detectors, Scorpio and Deckard have lower recalls. Notable is how consistent their recall is across the clone types. To prevent bias between recall measurements, the Mutation Framework uses the same original code fragments with each of the 15 mutation operators, and injects each of the 15 resulting clones at the same locations in the subject system. Therefore Scropio and Deckard may not have any deficiency for any particular clone type, but rather failed to detect these clones due to general deficiencies in its parser or detection algorithms.

Overall, there is strong agreement between our expected recall and the Mutation Framework's results. Agreement is found in 30 out of 33 Java cases, and 20 out of 27 C cases. In the cases of disagreement, the Mutation Framework consistently measured a lower recall. Strong agreement suggests confidence in the accuracy of the Mutation Framework. We did not notice any anomalies in the Mutation Framework's results.

G. Bellon's Framework vs Mutation Framework

In this section we directly compare the recall measurements of Bellon's Framework and the Mutation Framework. Since the two benchmarks were constructed differently (mined versus synthetic clones), we consider them to agree if the measured recalls are within 15%. Agreement between the frameworks is show in Table VII. Despite the differences in their approaches, it is reasonable to expect the frameworks to agree. The Mutation Framework tests the tools against a comprehensive (and empirically validated) taxonomy of the types of differences that can occur between clones. The base code fragments used for synthesis and injection locations in the subject system are randomly varied to ensure variety. We expect that if tools perform well for the Mutation Framework's synthesized clones, that this performance should transfer to clones naturally produced by developers.

However, in very few cases do these frameworks agree. They agree on the type 1 recall of CPD (Java), Duplo, iClones and Simian, as well as the type 2 recalls of CCFinderX (Java), CPD and Duplo. For CCFinderX (type 2, Java) and Simian (type 1, C), this agreement is only with Bellon's *ok* recall. We have shown that the *ok* recall can be unreliable. The frameworks agree in the cases where a tool does not support a particular clone type if either the *b-ok* or *good* recalls are considered. The frameworks disagree in some of these cases when the *ok* recall is used, which supports our findings that the *ok* metric can lead to incorrect recall measurements for clone types a tool doesn't support. In all other cases, the frameworks disagree on the tools' recall. Generally, the Mutation Framework measures a higher recall in these cases. With Bellon's Framework, the tools generally had lower recall for C clones, but this is not common in the Mutation Framework results.

Disagreement between the frameworks over NiCad and SimCad is not suspicious. These tools detect clones at the code block granularity: code that starts and ends with matched brackets, i.e., '{...}'. The Mutation Framework generates clones at this granularity to support more tools. Tools that search at a lower granularity (i.e., within code blocks) do not have a disadvantage with the Mutation Framework. However, NiCad and SimCad may fail to detect clones in Bellon's corpus that are much smaller than a code block. Despite this, NiCad has the top Java type 3 recall by Bellon's Framework.

It is particularly strange that the frameworks disagree for type 1 and type 2 recall. When the Mutation Framework measures a high recall for these types, it has certified that the tool can handle all 10 of the variations in type 1 and type 2 clones from the clone taxonomy. Many of the modern tools received this certification. As per mutation analysis, the Mutation Framework mutates only a single random difference into the reference clones. The tools detect type 1 and type 2 clones by removing or normalizing these differences during parsing or pre-processing steps. Therefore, the tools should have no problems detecting type 1 and type 2 clones, no matter the density of the type 1 and type 2 features. It is odd that Bellon's Framework measures considerably lower type 1 and 2 recall for some of the tools that have very high recalls by the Mutation Framework. It is possible that this is due to changes in clone detection preferences between the 2002 tools and the modern tools. Detection preferences may include clone granularity, scope, what constitutes a true positive, what clones are useful to report, and so on.

The frameworks do not agree on the type 3 recall of any of the type 3 tools, with the Mutation Framework consistently measuring a higher recall. The Mutation Framework shows that most of the type 3 detectors are able to handle the types of differences that can occur between type 3 clones. We constrained the Mutation Framework to generate clones with similarity no less than 70%. Bellon provided no specification for his type 3 clones. It may be that Bellon's type 3 clones contain a higher degree of dissimilarity, more than the tools allow, which would result in a lower recall from Bellon's framework. It is also possible that Bellon's corpus does not have sufficient type 3 representation to accurately measure recall. Bellon's corpus was built using tools contemporary to 2002, when type 3 detection was not as well developed. However, is is strange that the modern tools are not able to detect more of the type 3 clones found by their "outdated" predecessors. This suggests that type 3 preferences have changed, and the modern tools target a newer specification.

The Mutation Framework has a much stronger agreement with our expectations than Bellon's Framework, as shown in Table VI. The Mutation Framework agrees with our expectations in 90% (Java) and 74.1% (C) of the cases, while Bellon's Framework only agrees in 24.2-30.3% (Java) and 14.8-29.6% (C), depending on the metric used. We suspect that in the cases where neither tool agrees with our expectation, that

TABLE VII: Mutation Framework (MF) vs. Bellon's Framework (*ok*, *b-ok*, *good*) - ⊗ = Agree (±15%) ◯ = Disagree

Language	Tool	CCFX			ConQat			CPD			CtComp.			Deckard			Duplo			iClones			NiCad			Scorpio			SimCad			Simian			% Agree
	Clone Types	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	
Java	MF vs *ok*	◯	⊗	◯	◯	◯	◯	⊗	⊗	◯	◯	◯	◯	-	-	-	◯	◯	⊗	⊗	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	⊗	◯	⊗	23.3%
	MF vs *b-ok*	◯	◯	⊗	◯	◯	◯	⊗	⊗	⊗	◯	◯	⊗	-	-	-	⊗	⊗	⊗	⊗	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	⊗	◯	⊗	36.7%
	MF vs *good*	◯	◯	⊗	◯	◯	◯	◯	◯	⊗	◯	◯	⊗	-	-	-	⊗	⊗	⊗	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	⊗	◯	⊗	26.7%
C	MF vs *ok*	◯	◯	◯	-	-	-	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	⊗	⊗	⊗	◯	◯	◯	◯	◯	-	-	-	◯	◯	◯	⊗	◯	◯	14.8%
	MF vs *b-ok*	◯	◯	⊗	-	-	-	◯	⊗	⊗	◯	◯	⊗	◯	◯	◯	⊗	⊗	⊗	⊗	◯	◯	◯	◯	◯	-	-	-	◯	◯	◯	◯	◯	⊗	33.3%
	MF vs *good*	◯	◯	⊗	-	-	-	◯	⊗	⊗	◯	◯	⊗	◯	◯	◯	⊗	⊗	⊗	◯	◯	◯	◯	◯	◯	-	-	-	◯	◯	◯	◯	◯	⊗	29.6%

our expectation is incorrect. Bellon's strong disagreement with our expectation, and the suspicions we raise about its results, suggest that it is not accurate for modern tools. The Mutation Framework's strong agreement with out expectations suggest that it may be a good solution for evaluating the modern tools. In cases where the Mutation Framework disagrees with our expectations, we suspect that our expectation is incorrect, and the Mutation Framework accurate.

VIII. Threats to Validity

There are three primary threats to the validity of this study. (1) Our expectations of the tools' recall may not be accurate. We maximized our accuracy by consulting the tools' documentation, publication, literature surveys, and developers (when available). Furthermore, we allowed a ±12.5% range around our expectation to compensate for some inaccuracy. We used these expectation ranges as a baseline for our confidence in the benchmarks. (2) The tool configurations may not be optimal. We created targeted configurations by consulting the tools' defaults and documentation, which is how the average user would configure the tools for their use cases. While other configurations might give higher recall, our configurations measure the recall the average user can expect. (3) The Mutation Framework uses artificial clones. However, these clones are generated using mutation analysis, which is a well established technique in other fields including software testing. The clones are generated using a comprehensive clone taxonomy empirically validated against real clones [3], so the generated clones should be realistic.

IX. Conclusion and Future Work

In this paper, we compared the recall performance of eleven modern clone detection tools. We began by researching our expectations for these tools. We then evaluated the tools using different variants of Bellon's Framework, as well as the Mutation and Injection Framework. We found inconsistencies between our expectations and the results of Bellon's Framework. Bellon built his corpus by mining the output of tools contemporary to 2002. These clones and Bellon's procedures may not reflect the clone detection and reporting preferences of modern tools. Our findings suggest that Bellon's Framework may not be accurate for modern tools, and that an updated corpus may be warranted. We found agreement between our expectations and the results of the Mutation Framework. The Mutation Framework indicates that ConQat, iClones, NiCad and SimCad are very good options for detecting all three clone types. We believe the Mutation Framework could be a good solution for benchmarking modern clone detection tools. However, benchmarking with real data is also important. A priority of our future work is to update Bellon's Framework with clones detected by these tools.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queens University, Tech. Rep. TR 2007-541, 2007, 115 pp.

[2] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013.

[3] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. of Comput. Program.*, pp. 577–591, 2009.

[4] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in *WCRE*, 2003, pp. 285–294.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 577–591, 2007.

[6] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *IWSC*, 2013, pp. 8–9.

[7] H. Murakami, Y. Higo, and S. Kusumoto, "A dataset of clone references with gaps," in *MSR*, 2014, pp. 412–415.

[8] B. Baker, "Finding clones with dup: Analysis of an experiment," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 608–621, 2007.

[9] C. K. Roy and J. R. Cordy, "A mutation/ injection-based automatic framework for evaluating code clone detection tools," in *ICSTW*. IEEE, 2009, pp. 157–166.

[10] S. Bellon, "Vergleich von Techniken zur Erkennung duplizierten Quellcodes," Master's thesis, Universität Stuttgart, 2002, 156 pp.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *Softw. Eng., IEEE Trans. on*, vol. 28, no. 7, pp. 654–670, 2002.

[12] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *ICSE*, 2009, pp. 603–606.

[13] "Cpd," http://pmd.sourceforge.net/.

[14] W. Toomey, "Ctcompare: Code clone detection using hashed token sequences," in *IWSC*, 2012, pp. 92–93.

[15] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.

[16] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *ICSM*, 1999, pp. 109–118.

[17] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR*, 2009, pp. 219–228.

[18] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *ICPC*, 2011, pp. 219–220.

[19] Y. Higo and S. Kusumoto, "Enhancing quality of code clone detection with program dependency graph," in *WCRE*, 2009, pp. 315–316.

[20] M. Uddin, C. K. Roy, and K. A. Schneider, "Simcad: An extensible and faster clone detection tool for large scale software systems," in *ICPC*, 2013, pp. 236–238.

[21] "Simian," http://www.harukizaemon.com/simian/.

[22] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *ESEC/FSE*, 2013, pp. 455–465.