Benchmarks for Software Clone Detection: A Ten-Year Retrospective

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan, Canada
chanchal.roy@usask.ca

James R. Cordy
School of Computing
Queen's University, Canada
cordy@cs.queensu.ca

Abstract—**There have been a great many methods and tools proposed for software clone detection. While some work has been done on assessing and comparing performance of these tools, very little empirical evaluation has been done. In particular, accuracy measures such as precision and recall have only been roughly estimated, due both to problems in creating a validated clone benchmark against which tools can be compared, and to the manual effort required to hand check large numbers of candidate clones. In order to cope with this issue, over the last 10 years we have been working towards building cloning benchmarks for objectively evaluating clone detection tools. Beginning with our WCRE 2008 paper, where we conducted a modestly large empirical study with the NiCad clone detection tool, over the past ten years we have extended and grown our work to include several languages, much larger datasets, and model clones in languages such as Simulink. From a modest set of 15 C and Java systems comprising a total of 7 million lines in 2008, our work has progressed to a benchmark called BigCloneBench with eight million manually validated clone pairs in a large inter-project source dataset of more than 25,000 projects and 365 million lines of code. In this paper, we present a history and overview of software clone detection benchmarks, and review the steps of ourselves and others to come to this stage. We outline a future for clone detection benchmarks and hope to encourage researchers to both use existing benchmarks and to contribute to building the benchmarks of the future.**

I. Introduction

Software clones are defined to be duplicated or similar fragments of code in a software system [1], [2]. Copying a code fragment and reusing it by pasting with editing changes is a common practice in software development [3], [1], [4]. Developers in fact often intentionally practice cloning because of the underlying benefits, such as faster development, reuse of well-tested code, or even time limits assigned to them [1], [5]. As a result, in a given software system there typically could be 7% to 24% cloned code [6], and in some systems as much as 50% [7]. On the other hand, clones are the #1 code "bad smell" in Fowler's refactoring list [8]. Recent studies with both industrial and open source software show that while clones are not always harmful and can be useful in many ways [9], many of them can also be detrimental to software maintenance [9], [10], [11], [12]. Of course, reusing a fragment containing unknown bugs may result in fault propagation, and changes involving a cloned fragment may require changes to all fragments similar to it, multiplying the work to be done [1]. Additionally, inconsistent changes to the cloned fragments during any updating processes may lead to severe unexpected behaviour [11]. Clones are thus considered to be one of the primary contributors to the high maintenance cost of software, which is up to 80% of the total development cost [9].

The era of Big Data has introduced new applications for clone detection (e.g., [13]). For example, clone detection has been used to find similar mobile applications [14], to intelligently tag code snippets [15], to identify code examples [16], and so on. The dual role of clones in software development and maintenance, along with these many emerging new applications of clone detection, has led to a great many proposed clone detection tools. Some of those techniques are simply text-based (e.g., [17]), some are token-based (e.g., [18]), some are tree-based (e.g., [19], [20], [21]), some are metrics-based (e.g., [22]), some are graph-based [23], [24], and some are hybrid, for example parser-based but with text comparison (e.g, [25]). Scalability has become an issue, and the scalability of clone detection tools is now a major concern.

While these tools use a variety of different technologies and intermediate representations (e.g., text, token, tree, graph and so on), their one thing common is that they are all information retrieval (IR) [26] methods, since they are designed to detect and report similar fragments in software systems. The standard for evaluating performance of IR tools is the measurement of their precision and recall, and clone detection tools are no exception, particularly when dealing with large software systems. Precision measures how accurately the subject tool retrieves only *real* clones (true positives) in a software system, whereas recall measures how comprehensively the tool finds *all* of the clones in the system. For an ideal tool, both precision and recall should be high, since we would like to detect all the real clones. Precision can be measured by manually validating a statistically significant sample of the clones detected by a tool. Measuring recall on the other hand requires an oracle that knows of all the clones in a system or corpus, and evaluation of the subject tool's performance in detecting all of those known clones.

Unfortunately, when we started working in this area about ten years ago, despite a decade of active research there had been limited evaluation of clone detection tools in terms of precision and recall. In some cases precision was measured, and by only manually validating a small subset of randomly

978-1-5386-4969-5/18 © 2018 IEEE

SANER 2018, Campobasso, Italy
Retrospective Papers

selected clones detected by the subject tools [23], [24], [21]. Recall on the other hand had not been measured at all, possibly because of the challenges involved in creating an oracle. Although there had been some attempts at measuring precision and recall, along with time and space requirements, in a few tool comparison experiments [2], [1], [27], [28] and individual tool evaluations [29], [26], [25], they faced several challenges. First of all, it was challenging to create a large enough base of reliable reference data. Secondly, it was expensive to manually validate thousands of candidate clones. And thirdly, there was always the serious question of the reliability and subjectivity of the (human) judges.

Because of these challenges, experiments of the time either used no reference data [28], considered the union of some tools' results as an oracle by hand validating a small sample [2], [1], [27], [29], or detected clones in only a small software system and then manually validated the detected clones [30], [26]. Of course, the union results of the subject tools may still give good relative measurements of precision and recall of the subject tools [1]. However, as Baker [31] pointed out, when using the union results as the reference, there is no guarantee that even the whole set of participating tools actually detected all of the clones in the subject systems. Manually building an oracle for a subject system or manually validating a large sample of the detected clones is also challenging [31]. It took 77 hours for Bellon et al. to validate only 2% of the candidate clones in their tool comparison experiment [1]. Manually oracling a small system was possible in some cases [27], [26]. However, even when considering a relatively small system such as *Cook*, and even if we consider only function clones, we would need to examine all pairs of functions of that system, resulting in millions of function pairs to manually check. Even if we had the time available to do that, it would be impossible to do so without some level of inadvertent human error. There is also the question of the reliability and subjectivity of the judges, which neither these experiments nor the individual tool authors attempted to evaluate. This is a crucial issue in evaluating clone detection tools since even expert judges often disagree in creating clone reference data [32]. Another important aspect is the capability of dealing with different types of clones, in particular, Type 3 or "near-miss" clones, where there could be statement level differences between the cloned fragments, where statements could be added, modified or deleted from the reused pasted fragments. At the time, there were no tool evaluations except Bellon et al. and Falke et al. [29] that reported precision and recall values for different types of clones.

In this paper, we present a retrospective view of benchmarks for software clone detection tools over the last ten years. We begin with our empirical study on detecting function clones published at the 15th Working Conference on Reverse Engineering (WCRE 2008), analyze the impact of that study in creating benchmarks for clone detection, and then provide a road map of how we extended and scaled the techniques of that paper to one of the largest benchmarks in software clone detection research today, with more than eight million manually validated clone pairs in a large inter-project repository of 25,000 Java projects. We then conclude with our vision for creating even better benchmarks for software clone detection in the future.

The rest of this paper is organized as follows. We begin with clone detection and benchmark related terminology in Section II. We then provide a summary of our WCRE 2008 paper and its extended journal version, and demonstrate the overall impact of that work on cloning research and benchmarking in Section III. In Section IV we then provide how different cloning benchmarks evolved over time to its current form. We outline what we have learned about how benchmarks for clone detection should be created in Section V along with our suggestions for future research directions. Finally, Section VI concludes the paper.

II. Background

We begin with a basic introduction to clone detection and benchmarking terminology [1], [2].

A. Clones

Definition 1: Clone Types. There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location.

- *Textual Similarity*: We distinguish the following types of textually similar clones:
  - **Type 1:** Identical code fragments except for variations in whitespace, layout and comments.
  - **Type 2:** Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
  - **Type 3:** Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
- *Functional Similarity*: If the functionality of two code fragments is identical or similar, we call them *semantic* or *Type 4* clones.
  - **Type 4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

B. Clone Detection

Definition 2: Clone Pair. A pair of code portions/fragments is called a clone pair if there exists a clone-relation between them, i.e., a clone pair is a pair of code portions/fragments which are identical or similar to each other.

Definition 3: Clone Class. A clone class is the maximal set of code portions/fragments in which any two of the code portions/fragments hold a clone-relation, i.e., form a clone pair. A clone class is therefore, the union of all clone pairs which have code portions in common [7]. Clone classes are also known as clone groups, or clone communities [22].

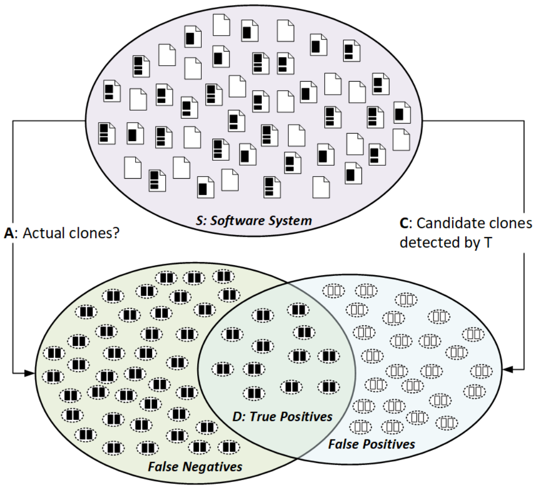

Fig. 1. Definitions for Precision and Recall in Clones

C. Benchmarking

Since clone detection is at its root an information retrieval (IR) problem, the performance of clone detection tools is typically measured using the standard IR based metrics, namely *precision* and *recall*. Precision reflects how well a clone detection tool detects actual clones from a subject system. It is the ratio of the actual clones detected by tool T from software system S with the total number of candidate clones reported (either true positives or false positives) by the tool, as shown in Eq. 1. A clone detector could falsely report two code fragments as a clone pair (false positives), and/or could indeed detect two code fragments as a clone pair that are actually similar (true positives). The tool could even fail to detect some actual clone pairs (false negatives) from the subject system. Figure 1 shows a system with many similar fragments (actual clones, indicated by solid fragments). Given an oracle for the actual clones A of a system S, and a set of candidate clones C detected by a given tool T, only the common set D are accurately detected clones.

Definition 4: Precision.

$$precision = \frac{|Actual\ clones\ in\ S\ detected\ by\ T\ (\mathbf{D})|}{|Candidate\ clones\ in\ S\ reported\ by\ T\ (\mathbf{C})|} \quad (1)$$

A tool should be sound enough that it detects only a small number of false positives, i.e., it should find duplicated code with high precision.

Definition 5: Recall.

$$recall = \frac{|Actual\ clones\ in\ S\ detected\ by\ T\ (\mathbf{D})|}{|All\ actual\ clones\ in\ system\ S\ (\mathbf{A})|} \quad (2)$$

A tool should be comprehensive enough that it detects the great majority (or even all) of the clones in a system, i.e., it should have a high level of recall.

Often, duplicated fragments are not directly textually similar, as editing activities on the copied fragments may disguise their similarity to the original. Nevertheless, a clone-relation may exist between them. A good clone detection tool will be robust enough in identifying such hidden clone relationships that it can detect most or even all the clones in a subject system. The recall measurement reflects this as the ratio of the actual clones detected by a tool T from a software system S to all of the actual clones of S (Eq. 2). A major challenge in the clone detection community is the lack of a reliable benchmark (i.e., a reliable set of actual clones A for the subject system S in Figure 1).

Fig. 2. Word cloud of the titles of articles citing the WCRE 2008 paper

III. The WCRE 2008 Paper and Its Extension: An Overview

Our work in clones began with a comprehensive survey of the clone detection literature in 2007 [1]. In that study we noticed that while there had already been a great deal of work in clone detection and analysis, there was a marked lack of work in near-miss clone detection, benchmarking and analysis of clone detection tools. As proposed in some earlier studies [24], we suspected that there might be more Type 3 near-miss clones in software systems than Types 1 and 2. Thus we first focused on developing a clone detection tool that could accurately detect near-miss clones. In particular, we proposed a new hybrid clone detection tool, NiCad [25], designed to combine the strengths of both text-based and parser-based clone detection techniques in order to overcome their limitations. In order to evaluate our new method, we then began exploring our second goal, the empirical evidence of near-miss clones, concentrating in particular on near-miss function clones in open source software, the subject of the WCRE 2008 paper and its extended journal version.

Using our new NiCad clone detection tool, we examined the clones of 15 open source C and Java systems, including the entire Linux Kernel and Apache httpd. Although NiCad was originally designed to allow extensive code normalizations for reducing gaps between fragments, flexible filtering for removing unnecessary code statements (e.g., declarations) and flexible pretty-printing, in the WCRE 2008 work we intentionally used the basic version of NiCad. We aimed at finding exact and near-miss intentional function clones, clones that appear because of copy/paste followed by adaptation to context by adding, deleting and/or modifying statements. We analyzed the use of cloned functions in those subject systems in several different dimensions, including language, clone size, clone location and clone density by proportion of cloned functions.

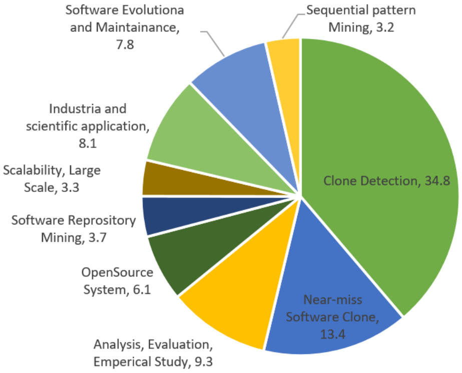

Fig. 3. Pie chart of the most frequently used words in articles citing the extended article

Because Bellon's experiment [2] was the most extensive to that point, we chose to analyze all of the C and Java systems of his experiment, including the systems used in his test runs, and added Apache *httpd*, *JHotDraw*, the entire Linux Kernel and a number of small systems. We manually validated all of the detected clones using a two-step process. First, we used NiCad's interactive HTML output of the detected clones to manually examine an overall view of each of the clone classes. Then we used NiCad's XML output to do a pairwise comparison of the original source of the cloned fragments of each of the reported clone pairs using Linux *diff*. This diff-based comparison helped us to relate the reported clone pairs to the textual similarity of the original source, and we manually checked all reported clone pairs with lower original text similarity. With the help of the interactive HTML view and the tool support for comparing original source, we were able to do the validation of all NiCad's reported clones in less than one man-month. In order for others to use the validated clones, we provided a complete catalogue of the detected clones in an online repository in a variety of formats.

Since its original publication, the original WCRE 2008 article has been widely cited in the cloning community and beyond, and it currently has more than 100 Google Scholar citations. In order to have a quick overview of the cited articles we extracted the titles of the papers and generated a word cloud (Figure 2). In order to better expose words related to the actual applications, we first removed high frequency but common words such as "software", "code" and "clones".

The paper has been widely cited in different areas of cloning research and beyond, such as clone detection [33], [34], [35], [36], [37], [38], [39], proactive cloning and their management [40], [41], [42], [43], clone analysis and empirical studies [12], [44], [45], [46], [47], [48], [49], [50], evaluation of clone detection tools [51], [52], [53], reusing of code in Android market [54], refactoring recommendations of clones [55], porting of scientific applications [56], code obfuscation [57], execution trace analysis [58], process improvement [59], porting OpenMP applications [60], fault handling in WS-BPEL [61] and so on.

In an invited special journal article [6], we then extended the original WCRE study with more open source systems in three different languages, C, Java and C#. As in the original WCRE 2008 study, we manually validated all the detected clones and provided a complete catalogue of those clones in an online repository in a variety of formats. The extended version has been widely cited as well. Figure 3 shows a pie chart of the most frequently used words in the titles of the papers citing the extended journal version.

IV. The Evolution of Clone Detection Benchmarks

A. A Short History

Benchmarks are essential to objectively evaluating and comparing clone detection tools. While the precision of a tool can be measured by manually validating a statistically significant sample of the detected clones, measuring recall is more challenging, requiring an oracle with knowledge of all of the clones in a system (see Section II). While there are many ways that such an oracle could be created, little was available in terms of clone oracles when we began our clone detection research in 2007. Despite a decade of active clone detection research, there was only one notable benchmark available, the Bellon et. al. benchmark [2]. This benchmark provided a great first start towards building cloning benchmarks for comparing and evaluating clone detection tools, and has been widely used over the years [1].

However, as noted in Section III there was a marked lack of representative Type 3 (near-miss) clones, which motivated us to conduct empirical studies for near-miss clones using our NiCad clone detection tool. A result of those studies was a set of thousands of hand-validated function clones (see Section III for details). Unfortunately, while they included many Type 3 near-miss clones, our reference data also suffered from limitations similar to Bellon's [31]. Similarly to Bellon, we used the opinion of existing clone detection tools (in our case NiCad) to provide the clones to be validated, and the hand validation was based solely on one individual's judgement (the first author of this paper).

Based partly on our observations in the WCRE 2008 experiment, in 2008 we proposed a fine-grained taxonomy of clone types based on editing operations [62] within which differences between tools and techniques could be better understood. We surveyed the literature to study the complete range of patterns of detected clones, and designed a covering set of editing operations to explain the creation of clones from originals. This taxonomy allowed us to explain the full range of possible intentional copy/paste clones as sequences of editing operations, and every Type 1,2, or 3 copy/paste clone can be explained as a combination of these operations.

This taxonomy opened up to us a new strategy for objective evaluation of clone detection tools. By treating the editing operations as mutation operators, we could generate any number of pre-validated clones in a system by automatically copying and pasting fragments of code, applying our editing operations to generate a clone, and injecting the clone back into the original system. Moreover, because we knew which editing operations had created the clone, we knew exactly what kind of clone (Type 1,2, or 3) each generated clone was.

Based on this idea, we designed a mutation/injection-based framework for automatically evaluating the recall of clone detection tools [52] and began experimenting with fine-grained evaluation of NiCad and other tools. This method proved very effective, and we conducted a number of experiments using the technique [63], [64], including extending it for model clone detectors using graphical edit operations [65].

While the mutation/injection method has been a great first step towards overcoming some of the limitations of existing benchmarks, the validated clones were artificially created, and while they are modelled after observed edit paths for real clones, there is no guarantee that the recall measurements provided by this framework reflect the true recall of clone detection tools in practice.

For this reason, there have been continuing efforts to create new benchmarks from naturally existing clones, including extending Bellon's benchmarks with validated Type 3 clones, and our own efforts to build large scale benchmarks. The largest of these is our benchmark *BigCloneBench* [66], which has more than eight million manually validated clone pairs created using a search-based approach. In the following we provide executive summaries of the most extensive current benchmarks, highlight their strengths and weaknesses, suggest ways to address these weaknesses, and outline how clone benchmarking research has evolved over time.

B. Bellon's Benchmark

Bellon et al. [2] were the first to propose a comprehensive benchmark for comparing and evaluating clone detection tools. In 2001-02, Bellon conducted a tool comparison experiment [67] with the help of six other clone detection researchers, each of whom had their own clone detection tool. The six participating tools were the text-based Duploc [68], [17], the token-based Dup [69] and CCFinder [18], the abstract syntax tree-based CloneDr [19], the metrics-based CLAN [22], [2], and the program dependency graph-based Duplix [24]. Bellon provided four C and four Java programs and the tool authors detected clones using their tools and returned the results to Bellon. A total of 325,935 clone pairs were submitted, and Bellon manually validated a random 2% of those clones to build a reference corpus. From this process, he obtained 4,319 known actual clones (according to his personal judgement) by manually examining 6,528 randomly selected clone pairs from those submitted, spending about 77 hours. To avoid tool bias, Bellon was unaware of which participating tools detected which clone pairs. The precision of the tools was then measured as the ratio of the validated actual clones of the corpus detected by a the tool to the entire set of candidate clones it reported. Recall was measured using all of the 4,319 validated actual clone pairs as oracle, and measuring what proportion of those were detected by each tool. Bellon also released all of the associated scripts and software for using the benchmark, so that the experiement could be repeated and the validated corpus used to evaluate new clone detection tools.

Bellon's has been one of the most widely used benchmarks, and there have been several extensions to it as well [70], [64]. There are however several inherent limitations of the Bellon benchmark. For example, in another experiment, one of the participating tools, Dup [31] was used to analyze the findings of Bellon's benchmark. It was found that there were significant inconsistencies for different types of clones, and that there may have been issues with the manual validation since the validation procedure was not exposed. It was noted that while Bellon et al. had done a great job, an update of the reference corpus was essential. There have been also empirical assessments of the Bellon benchmark. For example, Charpentier et al. [71] also found disagreements in the different types of clones when they were judged by multiple independent judges. In a more recent experiment [72], they also found that the judgements of non-experts could be potentially unreliable. Of course, this is no surprise, since subjective disagreement is an inherent problem in clone detection.

Even with the validated corpus of known clones, measuring precision for a new tool using the Bellon benchmark is not straightforward. Because the set of known clones was based only on those detected by the original set of six participating tools, precision is relative to their capabilities and thus only meaningful for that original six. To meaningfully measure the precision of a new clone detection tool, one needs to extend Bellon's benchmark with the new clones detected by that candidate tool, and hand validate a corresponding subset of those. Measuring recall for a new tool is also problematic using the Bellon framework. Since the reference corpus was built using clones from six clone detectors, they are in a more favourable position in the recall measurements than the new tool. In order to have relative recall between the tools, it is important to validate clones from the new tool and add those validated ones to the corpus as well, so that the new tool has some of its own representatives in the benchmark. This would possibly give better relative recall of the tools including the new one. In one of our recent studies [64] we showed that this is definitely the case, and a new tool is at a distinct disadvantage when using the Bellon framework. We also experienced challenges in validating clones from the new tool. Clearly the validation procedure of Bellon et al. should be followed, but that procedure is unfortunately both subjective and not well documented. This was also observed in Baker's analysis of Bellon's study [31]. Furthermore, the benchmark is inherently biased by the capabilities of the original participating tools, and for that reason the reference data have few or no representative of the types of clones that the participating tools could not detect - in particular, certain kinds of Type 3 clones.

In an effort to at least partly overcome some of the challenges and limitations of the Bellon benchmark, in 2009 we designed the mutation/injection-based framework for automatically evaluating clone detection tools that we describe next.

C. A Mutation/Injection-Based Framework

As explained in Section II, accurate evaluation of the precision and recall of a clone detection tool requires an oracle that can serve as a reference for the actual clones to be detected in a system. This is challenging for several reasons [31]:

1) Oracling a systems is effort intensive, since it involves manually validating all possible pairs of code fragments of the subject system (e.g., for a system with n fragments, one would need to validate $n(n-1)/2$ pairs, resulting in millions pairs to validate even for small systems),
2) Validating one or only a few systems may not be enough, since they may lack a sufficient number or variety of clones,
3) The validation procedure is itself a subjective process, and there are often disagreements [71], and
4) There is in fact no crisp definition of code clones, which can depend on the particular use case of a clone detection user.

Bellon's was the first real attempt to build a benchmark, but it suffers from all of these limitations, as discussed in Section IV-B. In fact, it is likely impossible to have a true clone oracle for any real system, because there will always be subjective bias. For this reason, in 2009 we proposed the idea of artificially generating realistic clones, using mutation analysis to build an artificial clone benchmark [73], [52].

Our mutation/injection-based framework maps our comprehensive taxonomy of clone editing operations [52], [74] to a set of code mutation operators, each of which applies a known editing operation to generate a new clone from a given code fragment. These artificial fragments are then injected into the code of the original system, yielding a new set of artificial clones of known clone type (i.e., Type 1, 2 or 3). The precision and recall of a clone detection tool can then be evaluated, for either all clones or each individual clone type, with respect to its ability to detect these known, pre-validated artificial clones.

This mutation-based framework was intended to overcome each of the challenges above. First, we attempted to overcome the vagueness in clone definition by proposing an editing taxonomy of clones, enumerating the different ways that a developer could intentionally copy, paste and adapt an original code fragment. The taxonomy essentially mimics the copy/paste/edit behaviour of developers in software development, which we can agree represents the creation of at least some of the clones that all tools are interested in (addressing challenges 1 and 4 above). Because it is automatable, the mutation/injection-based approach can be used to generate thousands of randomly artificial clone pairs and inject them into the subject system one at a time, providing thousands of pre-validated clone oracles for each of the pairs (addressing challenges 1 and 3). Furthermore, the procedure can be automatically repeated for any system, to build any number of artificial benchmarks automatically (addressing challenge 2).

To implement this strategy, the framework has two main phases, *Generation* and the *Evaluation*. In the Generation phase, we use a set of mutation operators based on the editing taxonomy to generate a set of cloned fragments using TXL [75]. A set of original code fragments is selected from a code base consisting of a collection of real systems. For each of these original code fragments, we generate a large set of mutated fragments using the mutation operators. Each original code fragment forms a clone pair with its mutated version. In this way, we create thousands of mutated clone pairs, which are then injected into the subject code base one at a time, generating thousands of mutated versions of the original code base each with at least one known clone pair (the injected one). These form the oracle for evaluating the subject clone detection tool(s) in the Evaluation phase. The current version supports only function and block granularity clones for the Java, C and C# languages. However, it can easily adapt to any new language just by adding a TXL-based lightweight grammar for the language.

While the mutation/injection-based framework does address the major issues to some extent, it does suffer from its own limitations. For example, while the editing taxonomy was derived from observed real clones and validated to cover all of the kinds of clones found in a small set of systems, it may not describe all of the kinds of clones that we may be interested in. In particular, it may not fully reflect real clones, and the code fragments randomly selected for clone generation may not be representative of those a developer may choose for cloning in real software development. Furthermore, it cannot generate Type 4 clones. Randomly applying editing mutations also may not generate a realistic distribution of different kinds of clones, which may vary widely depending on the programming language and application domain.

The current version of the framework has been used to evaluate several state of the art clone detectors [64]. However, there is no guarantee that all clone detectors could be validated using this framework. For example, the clone detectors that require fully compilable code may not work with the framework. The framework guarantees that the mutated and injected code is syntactically valid. However, it cannot guarantee that the modified source files will compile. Finally, combining editing mutations can be problematic, because we do not know how many changes can be made before a code fragment stops being a clone. For this reason, our evaluation framework limits each artificial clone to a single mutation (which does however have the advantage that we know which type of clone it is).

D. Adaptations of the Mutation/Injection Framework

The mutation/injection framework was designed to be extensible with plug-in custom mutation operators. Thus it is possible to design any sort of mutation operators for synthesizing any type of clone for recall (and possibly precision) measurements. For example, recently we adapted the framework for evaluating very large Type 3 gapped clone detectors [76]. By very large gaps we mean that after copying a code fragment a developer might insert a sequence of several new code statements to adapt to the new context. This sequence essentially creates a very large single gap in the pasted code fragment. We found that there are many such large gapped clones in software systems, and there have been no clone detectors for detecting such clones. We developed such a clone detector, *CCAligner* [76], and evaluated it with the extended mutation framework. The mutation/injection framework has also been successfully adapted to evaluating clone detectors in software product lines [77] and Simulink models [65].

In software development, particularly in large organizations, similar software products known as *software variants* can emerge in various ways. We have extended and adapted the original framework for generating artificial software variants with known similarities and differences in the ForkSim [77] tool. Using ForkSim, we can generate thousands of variants of a software product similar to the way that we make thousands of mutants in our mutation framework. These variants can be used not only to automatically evaluate the recall of product line detection tools but also to semi-automatically measure their precision. In fact, they can be used in any research on the detection, analysis, visualization and comprehension of code similarity among software variants. Similarly to our original framework, ForkSim suffers from the same limitations outlined in Section IV-C.

Given the increasing use of model-driven engineering in software development and maintenance, there have been several model clone detection techniques and tools proposed in the literature. As for source code clone detectors, there has been also a marked lack of objective evaluation of such tools. To help address this issue, based on an analogous editing taxonomy and mutation operators for Simulink models, Stephan et al. [78], [65] have proposed a mutation/injection-based framework for objectively and quantitatively evaluating and comparing Simulink model clone detectors. Like our original framework, this framework is subject to the limitations discussed in Section IV-C.

E. Adaptations of Bellon's Benchmark for Type-3 Clones

Murakami et al. [70] proposed an updated version of Bellon's benchmark including location information for gap lines. They argue that because it is missing such location information, the benchmark can incorrectly evaluate detection of some Type 3 clones. By adding location information for embedded gaps, they showed that the augmented dataset can more accurately evaluate Type 3 clone detectors. In the original Bellon dataset, code clones are represented by file name and the start and end line numbers of the code fragment. Murakami et al. added line numbers for gap lines to this data, and were able to show different and possibly better results for three Type 3 clone detectors, namely NiCad, Scorpio [79] and CDSW [80]. By ignoring the identified gaps in the evaluation, they observed that for NiCad the number of detected clones did not really change, but for CDSW, the number of clones detected increased in nearly half of the cases, while Scorpio produced more interesting results. They found that the number of detected Type 3 clones also varied with the subject system - for example, it increased for *netbeans* but decreased for some others.

In another experiment [64] with Murakami's extension, we found that ignoring gap lines had minimal impact on the tools' measured Type 3 recall. With only two exceptions, we found that participating tools' Type 3 recall showed an absolute change of no more than $\pm 1.5\%$ when ignoring gap lines as compared to Bellon's original metrics. Exceptions were CPD's Type 3 recall for Java (an absolute increase of 7.2% in *ok* results), and iClones's Type 3 recall for C (an increase of 3.7% in *good* results). Like Murakami, we observed that ignoring gap lines has little or no effect on NiCad's measured recall. However, we found different results for the Type 3 recall of Scorpio and CDSW, possibly due to differences in how Scopio's element-list results were converted to line numbers. Overall, Murakami et al.'s Type 3 extension of Bellon's benchmark had little effect on evaluation in our experiments, although it can be used to better evaluate clone detection tools that report gap lines in their reported clones.

F. Avoiding Subjective Bias in Benchmarks Using Real Clones

As noted in Section IV-B, one of the problems with hand validating clones in clone benchmarks such as Bellon's is the potential for subjective bias of the validator [31], [71], [1], [32]. While a mutation-based framework can partly overcome this by automatically generating clones, the generated clones are nevertheless artificial and may not be representative of real clones. In our recent studies we also found that in some cases artificial clone references actually had negative effects on accuracy measurement [64]. In an attempt to address both these issues, Yuki et al. [81] have proposed a reference corpus of real clones with less subjectivity by automatically mining intentional clones from the revision history of a software system. In particular, they identified cloned methods that have been removed by software developers in past revisions using the merged method refactoring. Such refactoring is an indication that the developers took action to merge the two methods, a strong indication that they may have been real clones. While there may be other clones in the revisions,since the developers chose not to refactor them, they may not be as important. Thus a clone benchmark capturing historically refactored clones may reflect the importance of those clones to software developers, and clone detection tools can be more realistically evaluated using such a benchmark.

Since subjectivity in clone benchmarking has been a major threat [31], [71], at least for the cases when the validators are not the developers of the chosen subject systems [72], Yuki et al.'s benchmark has potential. It has been objectively built from refactored clones observed in actual practice, and thus there can be no human subjectivity or biases involved, one of the biggest strengths of this kind of benchmark. However, there are concerns as well. For example, it is difficult to build a reasonably sized benchmark using this technique. Although they analyzed more than 15,000 revisions across three software systems, Yuki et al. were only able to find 19 such refactored clone pairs, which is far too few to meaningfully evaluate clone detectors. Evaluations using such a small benchmark are unlikely to be reliable. Moreover, the benchmark only captures those method clones that have been refactored. There may be many more clones in the systems that the developers are aware of but intentionally chose not to merge because refactoring was not possible or not needed [82], [3]. In addition, some human judgement is still necessary to validate the similarity between merged candidate instances and the removed pairs.

G. Benchmarks for Functionally Equivalent Clones

Since a majority of clones are not textually or even syntactically identical, there is also a need for clone benchmarks that can evaluate clone tools to detect other types of clones, including Type 4 (functionally equivalent) clones. Krutz and Le [83] have proposed a benchmark consisting of a set of clones created using a mixture of human and tool verification to provide high confidence in clone validation. They chose three open source software systems, Apache, Python and PostgreSQL, and randomly sampled a total of 1,536 function pairs. Then they recruited three clone experts and four students who have programming experience in validating function clone pairs. In order to have further confidence in the validation, they also made use of four clone detection tools, SimCad [84], NiCad [25], MeCC and CCCD. In total, they validated 66 clone pairs of different types (43 Type 2 clone pairs, 14 Type 3, and nine Type 4) to form a benchmark from the 1,536 function pairs. This seems to be one of the best ways of creating real clone benchmarks, since the authors selected the function pairs randomly and validated manually several different ways, including multiple clone judges. However, the procedure is both time consuming and challenging. 66 clone pairs is a very small sample, which may not be representative of the proportion and distribution of real clones of the different types in the software systems. Given the large amount of expert time required, the method is also difficult to scale to the thousands of clones required for a statistically significant benchmark. And finally, because they used a set of existing clone detection tools to assist in their judgements, like Bellon's, the benchmark could be skewed to the participating tools.

H. Automatic Approaches to Building Large Type 3 Benchmarks

To address the scale issue, there have been attempts to automatically build large Type 3 clone benchmarks. Lavoie and Merlo [85] consider that a good benchmark should have a precise definition of code clone, should be able to include large enough systems, and should be able to be built automatically. With these objectives in mind, they propose a Levenshtein distance metric to automatically build Type-3 clone benchmarks. In order to provide a precise definition, they used the Levenstein metric as ground truth, defining only those pairs with a normalized Levenstein distance of less than 0.3 to be true clones. In this way the authors attempted to avoid both the extensive human effort and the subjective bias introduced by manual validation. Since the method is automatic, it can scale to benchmarks built from systems with millions of lines of code. Furthermore, since the approach can investigate all possible pairs in a system, the precision of subject tools can be measured automatically as well. Of course, the major threat to this benchmark is its reliance on Levenstein distance to define ground truth. One could argue that Levenstein distance is simply another clone detection method, and thus not reliable. It also suffers from some of the challenges to other benchmarks outlined above. However, since in order to scale most clone detection tools do not use computations as costly as Levenstein distance, this large benchmark can be still useful for evaluating the recall and precision of clone detection tools, at least on a relative scale.

Tempero [86] automatically constructed a clone benchmark of over 1.3 million method level clone pairs from the 109 different open source Java systems in the *Qualitas Corpus*, totalling approximately 5.6 million LOC. A clone detection tool, CMCD [36], was used to validate the benchmark, without any manual evaluation. Like the Lavoie and Merlo work above, this benchmark depends on the opinion of one clone detection method as oracle, and thus has all the same drawbacks. Nevertheless, this very large corpus can still be useful in relative evaluation experiments for other tools.

I. Large Inter-project Benchmarks with Functionally Equivalent Clones: BigCloneBench and BigCloneEval

The era of Big Data has spawned both new challenges and new opportunities for clone detection, including searches for clones across thousands of software systems. Recently Big Data-based clone detection and search algorithms have been proposed, but we lack for corresponding new benchmarks suitable to evaluate such emerging techniques. We have recently shown that previous leading benchmarks, such as Bellon's, can not be appropriate for evaluating more modern clone detection tools [64]. Our own mutation/injection-based framework also has two major issues in the context of Big Data applications:

1) It only creates artificial clones, which may not reflect real clones, and
2) Since it re-analyzes a subject system (with only one injected clone pair each time) thousands of times, it may not scale to work with truly large inter-project systems.

Thus we have built (and are still building) the *BigCloneBench* [66], [90], [89], a large inter-project clone benchmark, which currently has more than eight million manually validated clone pairs in more than 25,000 Java projects (365 million LOC). In this work we have attempted to avoid one of the major challenges in building benchmarks, the use of clone detection tools themselves creating the benchmark for evaluating them. Instead of using clone detection tools, we use Big Data search techniques to mine the inter-project repository for clones of frequently used functionalities (e.g., bubble-sort, file-copy, and so on). We make use of search heuristics to automatically identify code snippets in the repository that may implement a target functionality. These candidate code fragments are then manually tagged by clone experts either as true or false positives for the target functionality. The benchmark is then populated with all of those true and false positive clones of different functionalities. Each of the clone pairs is further classified as Type 1, 2, 3 or 4 based on their syntactic similarity.

By construction *BigCloneBench* includes all four types of clones, including Type 4 clones. Since there remain disagreements on the minimum syntactic similarity of Type 3 clones, it has been a challenge to separate Type 3 and Type 4 clone pairs that implement the same functionality. In order to partially address this problem, we further classified Type 3/4 clone pairs into four different categories based on their syntactic similarity [66]:

1) *Very-Strongly Type-3*, with a syntactic similarity in the range 90% (inclusive) to 100% (exclusive),
2) *Strongly Type-3*, with syntactic similarity in the range 70-90%,
3) *Moderately Type-3*, with syntactic similarity between 50-70%, and
4) *Weakly Type-3/Type-4*, with syntactic similarity below 50%.

The current (second) version of the benchmark includes more than eight million clone pairs for 43 distinct functionalities. We have used the benchmark in several tool evaluation studies [64], [87], [88], including and evaluation of tools for large-scale clone detection [87], [88], [76]. We have released an evaluation framework, *BigCloneEval* [89], which fully automates the recall evaluation of clone detection tools using *BigCloneBench*. We also released experimental artifacts that enable the comparison of execution time and scalability [87], [88], [90].

This benchmark has a number of strengths. It has all four clone types, including fine-grained classifications of Type 3/4 clones, and it includes intra-project and inter-project clones scattered across more than 25,000 software systems. This benchmark can thus be used to evaluate both Big Data clone detectors and classical detectors, and all different kinds of clone detectors, including semantic (Type 4) clone detectors. It does not use any clone detection tools in building the benchmark, so the tool-specific bias has been overcome in this benchmark. It also partially avoids the subjective bias of clone validation, since the validation process is indirect and guided by strict specifications. However, there are still limitations to this benchmark. For example, while subjectivity is minimized, the ultimate decision of whether two code fragments form a clone pair is still decided by human judges, and different judges may have different opinions, in particular because we have a wide range of similarities. Further, the clone candidates are limited to the 43 distinct functionalities we reported, which may or may not be representative of all cloned functionalities.

V. Discussion and Future Research Directions

As we've seen in the previous section, there are many ways one can build a clone detection benchmark, and the limitations and strengths of each benchmark depend on how it was created. There are also many challenges involved in building a benchmark. One of the most popular and widely used methods of creating benchmarks is to use the clone detection tools themselves as oracles, as was done in Bellon's original benchmark. This is one of the easiest ways to create a benchmark, since one can simply union the results of the tools and consider the unioned results as an oracle, but it has a number of problems. The participating tools are presumed to have perfect precision, and thus false positive clones can be added to the benchmark, which can hurt the recall of the subject clone detection tools during the evaluation. The approach can also be missing many true clones that none of the participating tools were able to detect. For this reason, this kind of benchmark is not suitable for measuring precision, because the subject tool may detect some actual clones that the benchmark considers to be false positives. Of course, Bellon took several measures to at least partially mitigate these issues (Section IV-B) but many of these challenges remain, as outlined by Baker [31] and others, including our own recent studies [64], [72].

Injection of either natural or artificial known clones into software systems is another way of building clone benchmarks. It is also possible to intentionally create new clones in software systems and use those new clones for benchmarking. Instead of mining the clones using some means (e.g., using the clone detectors), one can create and inject new clones and thus can make as many clones and types of clones as they want. However, as also noted by Bellon et al., when done by hand this process is very effort intensive and can only be used to make small benchmarks. Instead of manual injection, one can artificially create clones and inject them into the systems. This however has the problem that the clones are not real, and if too many artificial clones are injected into the systems, inaccurate evaluation can result. In our mutation/injection-based framework [63], [73], [52], we attempted to overcome this using a mutation-based technology, injecting one clone at a time into the software system and attempting to detect that single clone pair using the subject tool, randomly repeating the process thousands of times. However, as noted in Section IV-C, this process has several limitations, including the lack of real clones and potential performance problems when injecting clones into very large systems. Of course, this limitation depends on whether the subject tool is fast enough to rapidly detect clones in a large code base.

Validation by manually inspecting all the possible clone pairs of a subject system is possibly the most obvious method. One could consider examining each and every possible clone pair in a system and add the validated clones to the benchmark. Since subjective bias is an issue, multiple clone experts can also be used to validate the same clone pairs. However, as noted earlier, this method is very effort intensive since even for a small system which may only have 1000 code fragments of a target granularity (e.g., functions), one would need to manually verify (1000*999)/2, or about half a million clone pairs. One possible solution to this is to select a statistically significant random sample of possible clone pairs and then verify them with multiple experts, as is done by Krutz and Le [83]. However, one may not find enough true clone pairs in the randomly selected set. For example, Krutz and Le found only 66 clone pairs in their experiment. Another promising solution could be to use a functionality-based search heuristic that is distinct from clone detectors as we have done for *BigCloneBench*, and to manually verify all the detected candidates. Using a search targeting similar intended functionalities, the validation process is greatly simplified. However, it still has some of the common problems, as discussed in Section IV-I.

To build future benchmarks, one could possibly begin with combinations of these approaches, based on their context. For example, one could begin with Bellon's approach of manually validating a significant sample, and possibly address the subjective bias issue by having multiple judges, using tool support in the manual validation process as suggested by Baker. Making use of Levenshtein distance [85] in the manual validation process may greatly improve efficiency and thus more clones could be validated error-free. Since each of the benchmarks has its own strengths and weaknesses, one could combine the different methods to build a benchmark on the same dataset. For example, we could use the functionality-based search heuristics approach combined with an automated means of validation. We have been exploring for example using machine learning approaches [91], [92] to validate clones. Once a large sample of clone pairs from a variety of domains has been manually validated (as in *BigCloneBench*), one could apply machine learning techniques to learn their features and develop models for automatic classification of new clone pairs. If this process is successful and can be made scalable, a large sample of clone pairs could be validated automatically using a small fraction of the required human validation.

We also need to consider several important features of an ideal benchmark. First and foremost, a benchmark should be publicly available, and preferably free. This is essential not only for adaptation of the benchmark to other languages and systems, but also for repeatability and extension of evaluation experiments. Another major requirement is that a benchmark should be extensible by the community. While an ideal benchmark would not need such extensions, in practice such a perfect general benchmark is impossible to create. For this reason, it is also essential that the benchmark authors provide good documentation and use scenarios, and demonstrate the evaluation procedure with multiple clone detection tools. Of course, the benchmarks themselves should also be evaluated in order to have confidence on their accuracy. As discussed above, for various reasons most benchmarks are possibly not accurate. It is also important that a benchmark contain multiple software systems of different languages, varying system sizes including ultra large repositories, and different application domains. A good benchmark should have clone representatives of all four clone types, and in particular includes clones of Types 3 and 4. Clones should be validated using a repeatable standard procedure, either fully manually with one or more experts, or using a combination of experts and tool support, or using an artificial means that offers high confidence. Another important thing to consider is the use case scenario of the benchmark. In clone detection the right answer depends a lot on the intended use of the results, and there is a need for task-specific benchmarks which can discriminate between tools designed for different use cases. Benchmarks aimed at specific scenarios of use, for example refactoring, can also be more easily validated. Of course, it is important that a benchmark contains different types, sizes, granularities (e.g., arbitrary statements, block, functions and so on) and functionalities of clones. For example, in *BigCloneBench*, we we have millions of clones for more than 40 distinct functionalities from thousands of subject systems, containing a wide variety of clones of all four clone types. A wide range of clones is essential for accurately measuring the recall of a clone detection tool. Finally, there should be a large number of validated clone pairs (possibly also clone classes), preferably in the thousands or more, in order to guarantee statistical significance of the evaluation results. Some further details can be found elsewhere as well [93].

VI. Conclusion

When we began our work on benchmarking for code clone detectors with the original WCRE 2008 paper ten years ago, few resources were available to support objective evaluation of clone detection tools. While Bellon's original benchmark represented a good first effort, it was clear that there were many drawbacks to a benchmark based on the tools themselves, making it difficult to rely on the results of evaluations using it. When we set out to create alternatives that could be used to objectively evaluate the precision and recall of clone detection tools, little did we know the extent of the challenges to be faced. In this paper we have outlined those challenges and reviewed the past decade of work by ourselves and others to try to address them, highlighting our own path from small sets of hand validated clones, to injection of large sets of artificially generated mutants, and finally to the huge corpus of validated real clones in *BigCloneBench*. We outlined the many challenges that remain to be faced, and summarized our ideas on goals for future work in hopes that more efforts can be made to set clone detector evaluation and comparison on a solid objective footing.

Acknowledgments

The authors would like to thank Jeffrey Svajlenko for his extensive work in building cloning benchmarks, Khaled Saifullah and Rayhan Ferdous for some help in preparing this document. The authors would also like to thank the many researchers in cloning community who have helped advance the area of clone detection benchmarking. Their work has inspired and challenged us over these many years. Finally, the authors wish to thank the SANER 2018 Most Influential Paper award committee for recognizing our WCRE 2008 paper on empirical studies of clones with this honour.

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), by an Ontario Research Fund Research Excellence (ORF-RE) grant, and by a Canada First Research Excellence Fund (CFREF) grant coordinated by the Global Institute for Food Security (GIFS).

References

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

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

[3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *ESEC/SIGSOFT*, 2005, pp. 187–196.

[4] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *ISESE*. IEEE, 2004, pp. 83–92.

[5] I. D. Baxter, M. Conradt, J. R. Cordy, and R. Koschke, "Software clone management towards industrial application (Dagstuhl seminar 12071)," *Dagstuhl Reports*, vol. 2, no. 2, pp. 21–57, 2012.

[6] C. K. Roy and J. R.Cordy, "Near-miss function clones in open source software: an empirical study," *J. of Softw. Evolution and Process*, vol. 22, no. 3, pp. 165–189, 2010.

[7] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *WCRE*, 2004, pp. 100–109.

[8] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[9] C. Kapser and M. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008.

[10] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, "Simultaneous modification support based on code clone analysis," in *APSEC*, 2007, pp. 262–269.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE*, 2009, pp. 485–495.

[12] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *ACM Applied Comp. Review*, vol. 12, no. 3, pp. 20–36, 2012.

[13] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *FSE*, 2010, pp. 167–176.

[14] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014, pp. 175–186.

[15] J.-w. Park, M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Surfacing code in the dark: an instant clone search approach," *Knowledge and Information Systems*, pp. 1–33, 2013.

[16] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *ICSE*, 2014, pp. 664–675.

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

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

[19] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, 1998, pp. 368–377.

[20] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *WCRE*, 2006, pp. 253–262.

[21] 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.

[22] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM*, 1996, pp. 244 –253.

[23] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.

[24] J. Krinke, "Identifying similar code with program dependence graphs," in *WCRE*, 2001, pp. 301 –309.

[25] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, 2008, pp. 172–181.

[26] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *WCRE*, 1997, pp. 44 –54.

[27] J. Bailey and E. Burd, "Evaluating clone detection tools for use during preventative maintenance," in *SCAM*, 2002, pp. 36 – 43.

[28] V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *ASE*, 2004, pp. 336 – 339.

[29] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, pp. 601–643, 2008.

[30] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 804–818, 2005.

[31] B. S. Baker, "Finding clones with Dup: Analysis of an experiment," *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 608–621, 2007.

[32] 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.

[33] S. Grant and J. R. Cordy, "Vector space analysis of software clones," in *ICPC*, 2009, pp. 233–237.

[34] C. K. Roy and J. R. Cordy, "Are scripting languages really different?" in *IWSC*, 2010, pp. 17–24.

[35] S. K. Abd-El-Hafiz, "A metrics-based data mining approach for software clone detection," in *COMPSAC*, 2012, pp. 35–41.

[36] Y. Yuan and Y. Guo, "CMCD: Count matrix based code clone detection," in *APSEC*, 2011, pp. 250–257.

[37] E. Merlo and T. Lavoie, "Computing structural types of clone syntactic blocks," in *WCRE*, 2009, pp. 274–278.

[38] A. El-Matarawy, M. El-Ramly, and R. Bahgat, "Code clone detection using sequential pattern mining," *International Journal of Computer Applications*, vol. 127, no. 2, 2015.

[39] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, pp. 130–142, 2018.

[40] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can I clone this piece of code here?" in *ASE*, 2012, pp. 170–179.

[41] E. Juergens, "Why and how to control cloning in software artifacts," Ph.D. dissertation, Technische Universität München, 2011.

[42] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Predicting consistency-maintenance requirement of code clonesat copy-and-paste time," *IEEE Tran. on Soft. Engg.*, vol. 40, no. 8, pp. 773–794, 2014.

[43] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *ICSE*, 2017, pp. 665–676.

[44] W. T. Cheung, S. Ryu, and S. Kim, "Development nature matters: An empirical study of code clones in Javascript applications," *Empirical Software Engineering*, vol. 21, no. 2, pp. 517–564, 2016.

[45] M. Asaduzzaman, "Visualization and analysis of software clones," Master Thesis, University of Saskatchewan, 109 pp, 2012.

[46] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Science of Computer Programming*, vol. 95, pp. 445–468, 2014.

[47] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *MSR*, 2016, pp. 362–373.

[48] E. Merlo, T. Lavoie, P. Potvin, and P. Busnel, "Large scale multi-language clone analysis in a telecommunication industrial setting," in *IWSC*, 2013, pp. 69–75.

[49] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and forecasting near-miss clones in evolving software: An empirical study," in *ICECCS*, 2011, pp. 295–304.

[50] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 175–190, 2010.

[51] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in *ICSME*, 2015, pp. 131–140.

[52] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *ICST Mutation Workshop*, 2009, pp. 57–166.

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

[54] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the Android market," in *ICPC*, 2012, pp. 113–122.

[55] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *ICSME*, 2014, pp. 331–340.

[56] W. Ding, C.-H. Hsu, O. Hernandez, B. Chapman, and R. Graham, "Klonos: Similarity-based planning tool support for porting scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1072–1088, 2013.

[57] S. Schulze and D. Meyer, "On the robustness of clone detection to code obfuscation," in *IWSC*, 2013, pp. 62–68.

[58] L. L. Silva, K. R. Paixao, S. de Amo, and M. de Almeida Maia, "On the use of execution trace alignment for driving perfective changes," in *CSMR*, 2011, pp. 221–230.

[59] A. Imazato, Y. Sasaki, Y. Higo, and S. Kusumoto, "Improving process of source code modification focusing on repeated code," in *PROFES*, 2013, pp. 298–312.

[60] W. Ding, O. Hernandez, and B. Chapman, "A similarity-based analysis tool for porting OpenMP applications," in *Facing the Multicore-Challenge III*. Springer, 2013, pp. 13–24.

[61] A. Kocbek and M. B. Juric, "Towards a reusable fault handling in WS-BPEL," *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 02, pp. 243–267, 2014.

[62] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *ICPC*, 2008, pp. 153–162.

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

[64] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*, 2014, p. 10.

[65] M. Stephan, M. H. Alalfi, A. Stevenson, and J. R. Cordy, "Using mutation analysis for a model-clone detector comparison framework," in *ICSE*, 2013, pp. 1261–1264.

[66] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *ICSME*, 2014, pp. 476–480.

[67] S. Bellon, "Stefan bellon's clone detector benchmark," http://www.softwareclones.org/research-data.php.

[68] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37–58, 2006.

[69] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE*, 1995, pp. 86–95.

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

[71] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of Bellon's clone benchmark," in *EASE*, 2015, pp. 20:1–20:10.

[72] A. Charpentier, J.-R. Falleri, F. Morandat, E. Ben Hadj Yahia, and L. Réveillère, "Raters' reliability in clone benchmarks construction," *Empirical Software Engineering*, vol. 22, no. 1, pp. 235–258, 2017.

[73] C. K. Roy and J. R. Cordy, "Towards a mutation-based automatic framework for evaluating code clone detection tools," in *C3S2E*, 2008, pp. 137–140.

[74] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, pp. 470–495, 2009.

[75] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, 2006, pp. 190–210.

[76] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector," in *ICSE*, 2018, p. 12 pp. (to appear).

[77] J. Svajlenko, C. K. Roy, and S. Duszynski, "Forksim: Generating software forks for evaluating cross-project similarity analysis tools," in *SCAM*, 2013, pp. 37–42.

[78] M. Stephan, "Model clone detector evaluation using mutation analysis," in *ICSME*, 2014, pp. 633–638.

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

[80] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Gapped code clone detection with lightweight source code analysis," in *ICPC*, pp. 93–102.

[81] Y. Yuki, Y. Higo, K. Hotta, and S. Kusumoto, "Generating clone references with less human subjectivity," in *ICPC*, 2016, pp. 1–4.

[82] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic identification of important clones for refactoring and tracking," in *SCAM*, 2014, pp. 11–20.

[83] D. E. Krutz and W. Le, "A code clone oracle," in *MSR*, 2014, pp. 388–391.

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

[85] T. Lavoie and E. Merlo, "Automated type-3 clone oracle using Levenshtein metric," in *IWSC*, 2011, pp. 34–40.

[86] E. Tempero, "Towards a curated collection of code clones," in *IWSC*, 2013, pp. 53–59.

[87] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in *ICSE*, 2016, pp. 1157–1168.

[88] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with CloneWorks," in *ICSE*, 2017, pp. 27–30.

[89] ——, "BigCloneEval: A clone detection tool evaluation framework with BigCloneBench," in *ICSME*, 2016, pp. 596-600.

[90] J. Svajlenko and C. Roy, "The BigCloneBench tool page:," http://www.jeff.svajlenko.com/bigclonebench.html.

[91] J. Svajlenko and C. K. Roy, "Efficiently Measuring an Accurate and Generalized Clone Detection Precision using Clone Clustering," in *SEKE*, 2016, pp. 426–433.

[92] J. Svajlenko and C. K. Roy, "A Machine Learning Based Approach for Evaluating Clone Detection Tools for a Generalized and Accurate Precision," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 09–10, 2016, pp. 1399–1429.

[93] C. K. Roy, M. F. Zibran, and R. Koschke, "The Vision of Software Clone Management: Past, Present and Future (keynote paper)", in *CSMR-WCRE*, 2014, pp. 18-33.