The Vision of Software Clone Management: Past, Present, and Future (Keynote Paper)

Chanchal K. Roy Minhaz F. Zibran Rainer Koschke†
University of Saskatchewan, Canada †University of Bremen, Germany
{chanchal.roy, minhaz.zibran}@usask.ca, koschke@informatik.uni-bremen.de

***Abstract*—Duplicated code or code clones are a kind of code smell that have both positive and negative impacts on the development and maintenance of software systems. Software clone research in the past mostly focused on the detection and analysis of code clones, while research in recent years extends to the whole spectrum of clone management. In the last decade, three surveys appeared in the literature, which cover the detection, analysis, and evolutionary characteristics of code clones. This paper presents a comprehensive survey on the state of the art in clone management, with in-depth investigation of clone management activities (e.g., tracing, refactoring, cost-benefit analysis) beyond the detection and analysis. This is the first survey on clone management, where we point to the achievements so far, and reveal avenues for further research necessary towards an integrated clone management system. We believe that we have done a good job in surveying the area of clone management and that this work may serve as a roadmap for future research in the area**

***Index Terms*—Code Clones, Clone Analysis, Clone Management, Future Research Directions**

I. Introduction and Motivation

Copying existing code and pasting it in somewhere else followed by minor or major edits is a common practice that developers adopt to increase productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments residing in the code base. Those duplicate or near-duplicate code segments are commonly known as code clones. There are many reasons why developers intentionally perform such code cloning. Obvious reasons include reuse of existing implementations without "re-inventing the wheel". More comprehensive discussions on the reasons for code cloning can be found elsewhere [104]. Code clones may also appear in the code base without the awareness of the developers. Such unintentional/accidental clones may be introduced, for example, due to the use of certain design patterns, use of certain APIs to accomplish similar programming tasks, or coding conventions imposed by the organization.

The reuse mechanism by code cloning offers some benefits. For instance, cloning of existing code that is already known to be flawless, might save the developers from probable mistakes they might have made if they had to implement the same from scratch. It also saves time and effort in devising the logic and typing the corresponding textual code. Code cloning may also help in decoupling classes or components and facilitate independent evolution of similar feature implementations.

On the other end of the spectrum, code clones may also be detrimental in many cases. Obviously, redundant code may inflate the code base and may increase resource requirements. This may be crucial for embedded systems and systems such as hand held devices, telecommunication switches, and small sensor systems. Moreover, cloning a code snippet that contains any unknown fault may result in propagation of that fault to all copies of the faulty fragment. From the maintenance perspective, a change in one code segment may necessitate consistent changes in all clones of that fragment. Any inconsistency may introduce bugs or vulnerabilities in the system. Fowler et al. [35] recognize code clones as a serious kind of code smell.

However, during the software development process, duplication cannot be avoided at times. For example, duplication may be enforced by the limitation of the programming language's necessary mechanism to implement an efficient generic solution of a problem at hand. Code generators may also generate duplicated code that the developers may have to modify.

Although controversial, previous research reports empirical evidences that a significant portion (generally 9%-17% [145]) of a typical software system consists of cloned code, and the proportion of code clones in the code base may be as low as 5% [104] and as high as even 50% [103]. Indeed, due to the negative impact of code clones in the maintenance effort, one might want to remove code clones by active refactoring, wherever feasible. However, in reality, aggressive refactoring of code clones appears not to be a very good idea [22], and not all clones are really removable through refactoring. Due to the dual role of code clones in the development and maintenance of software systems, as well as the pragmatic difficulty in avoiding or removing those, researchers and practitioners have agreed that code clones should be detected and managed efficiently [95, 143].

Since the emergence of software clones as a research area in early 1990s, significant contributions over years made the field grow and become quite a mature area of research. Over the entire course of software clone research there have been only two notable general surveys on clones. Koschke [80], in 2007, presented a brief summary of the important findings about different aspects of software clones including cause-effect of cloning, clone avoidance, detection, and evolution along with a set of open questions. In the same year, Roy and Cordy [104] also published another survey containing a thorough review on those same areas with specific focus on clone detection tools and techniques. A few recent surveys either focus on detection [102, 107] or evolution of clones [99]. In this vision

978-1-4799-3752-3/14/$31.00 © 2014 IEEE

paper, we provide an extensive survey on code clone research with strong emphasis on clone management and point readers to future research directions.

This paper is organized as follows. In Section II, we present a systematic review on a repository of 353 publications appeared over 20 years. The review draws a “birds-eye" view on the overall contributions and growth along different dimensions of software clone research. This survey is the outcome of careful investigation of literature beyond the said repository (described in Section II), and through analysis in the light of our experience. Section III introduces different aspects of clone management activities starting with the definition and types of clones. While in Section IV, we list different stand-alone clone detection techniques, we discuss the IDE-based clone detectors in Section V. We then talk about clone documentation in Section VI and that of tracking over evolution in Section VII. We discuss clone evolution studies including visualization of clone evolution in Section VIII. Then, in Section IX, we discuss clone annotation. Section X presents the techniques for clone removal or clone based reengineering. In Section XI, we describe the analyses for the identification of potential clones as candidates for refactoring/reengineering including the visualization of clones, cost-benefit analysis and scheduling of clones for refactoring. In section XII, we briefly summarize the root causes for clones followed by clone management strategies in Section XIII. In Section XIV, we briefly describe the design space for a clone management system. Our view on the challenges for industrial adoption of clone management is presented in Section XV. Finally, Section XVI concludes the paper with a rough summary of the state of the art along with future research directions.

II. A Systematic Review of Clone Literature

There has been more than a decade of research in the field of software clones. To understand the growth and trends in the different dimensions of clone research, we carried out a quantitative review on related publications. Robert Tiras has been maintaining a repository [120] of scholarly articles that make significant contributions in the area. Until today, the corpus consists of 353 scholarly articles published between 1994 and 2013 in different refereed venues including Ph.D., M.Sc., and Diploma theses. The repository organizes the publications by categorizing them based on their contributions in four major sub-areas of clone research. The categories are as follows:

Detection Publications in this category address techniques and tools for the detection of software clones.

Analysis This category contains publications that perform analysis on the various traits of software clones, their etiology, existence, effects in software systems, as well as investigation of clone reengineering opportunities and implications. A majority of such publications report findings from qualitative or quantitative empirical studies.

Management Publications in this category address the issues, techniques and tools for the management of code clones beyond detection.

Tool Evaluation This category comprises the publications that contribute to the quantitative or qualitative evaluation of the techniques and tools for clone detection.

Figure 1 plots the number of distinct authors contributing to clone research in the years from 1994 through 2013. As the figure indicates, the clone research community has experienced a significant growth over the recent years. In Figure 2, we present the number of publications appear every year contributing to each of the four sub-areas of clone research. As seen in the figure, early work on software clone research was dominated by the research on clone detection with some work on analysis. In the recent years, the work on clone analysis and detection has grown significantly while clone management has emerged and growing as a significant research topic. Despite the fast growth of the clone research community, the work purely on clone management received relatively less attention compared to analysis and detection, which can be more clearly perceived from Figure 3. This, in combination with the realized importance of research in clone management, points to the further need and potential for research in this sub-area.

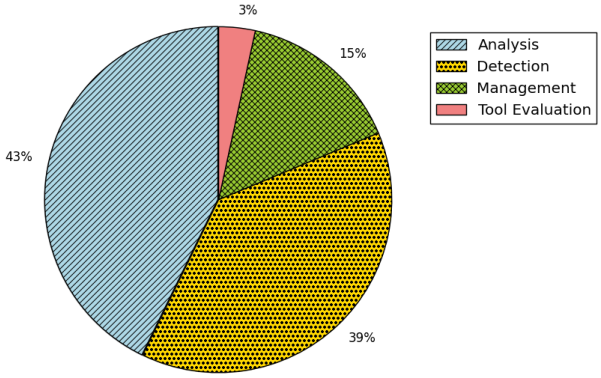

Fig. 3. Proportion of publications in each category over the period 1994–2013

It can also be noticed from both Figure 2 and Figure 3 that over the entire span (1994–2013) of software clone research a very few studies focused on the evaluation of clone detection techniques or tools, although more than 40 different clone detection tools have been produced realizing a wide variety of techniques [107]. Indeed, the detection of clones is a fundamental topic for software clone research, and the effectiveness of clone management largely depends on clone detection.

III. Clone Management

“Clone management summarizes all process activities which are targeted at detecting, avoiding or removing clones" [37]. Thus, clone management encompasses a wide range of categories of activities including clone detection, tracking of clone evolution, and refactoring of code clones. As support for these operations, the documentation and analysis of code clones can be regarded as parts of clone management. Moreover, clone visualization may also be an effective aid to clone analysis, and thus to clone management.

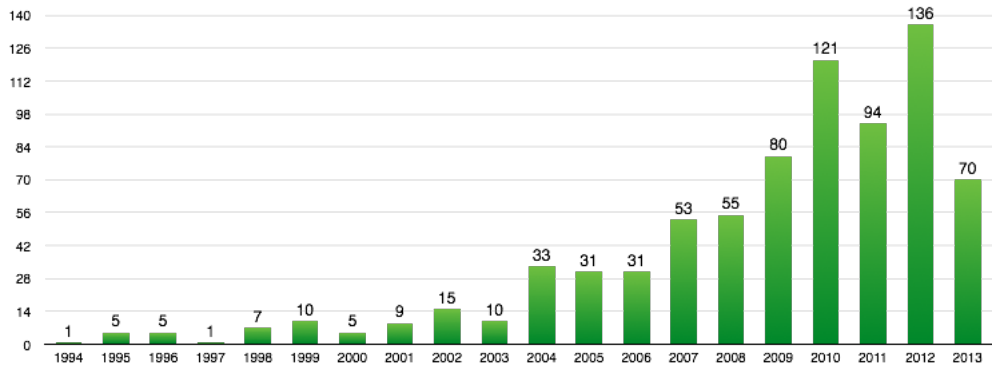

Fig. 1. Yearly number of distinct authors contributing to clone research

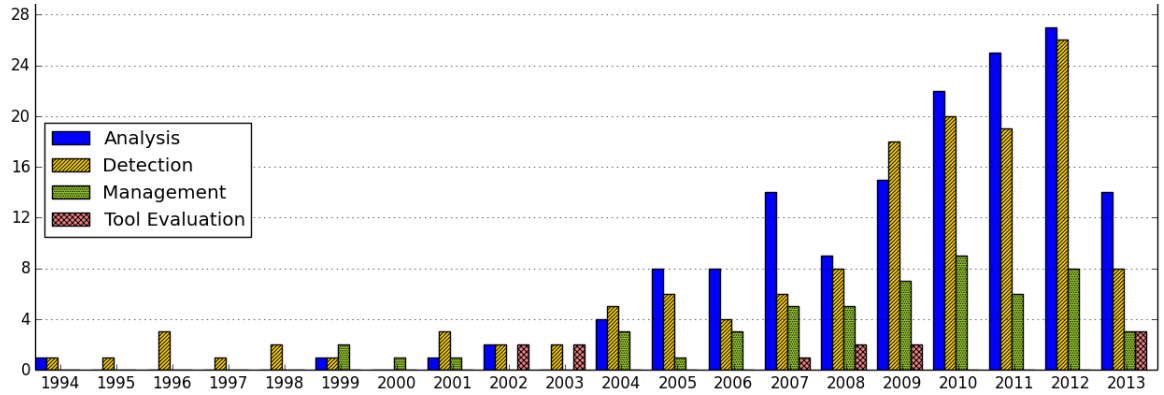

Fig. 2. Categories of publications on software clone research in different years

A. Definition of Code Clone

Though duplicate or similar code fragments are roughly known to be code clones, the definition of clone has remained more or less vague over the last decade. The vagueness is reflected in the definition given by Ira Baxter, "Clones are segments of code that are similar according to some definition of similarity" [12]. Despite ongoing debates in the research community, there is no consensus on a precise definition yet. Currently, a researcher's definition of similarity is typically constrained by the program representation and detection mechanism of his or her particular clone detector and, hence, varies from tool to tool and also from parameter settings controlling a tool. The least common denominator widely accepted today is the following taxonomy, which was created in the context of a study on comparing clone detectors [15]:

Type-1 Clone Identical code fragments except for variations in white-spaces and comments are *Type-1* clones.

Type-2 Clone Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments are called *Type-2* clones.

Type-3 Clone Code fragments that exhibit similarity as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones.

Type-4 Clone Code fragments that exhibit identical functional behaviour but are implemented through very different syntactic structures are known as *Type-4* clones.

Type-2 and type-3 clones are often collectively called *near-miss clones*. There have been alternative, more elaborated taxonomies proposed by Mayrand et al. [89], Balazinska et al. [7], and Kontogiannis [79], but they are not as widely used as the simple categorization by Bellon et al. [15].

A common definition is needed when empirical results are to be compared, for instance, on effects of clones or on accuracy of clone detectors. The difficulty to reach a consensus on a suitable definition, however, inevitably depends also on the purpose of the clone detection. A definition of similarity will include the "value" of a clone for the given task (e.g., bug fixing or refactoring). We do not foresee the advent of a unified definition, we rather expect that task-specific taxonomies of code similarity will emerge in the future and studies will further differentiate contexts and purposes of clones.

Ongoing research also attempts to deal with clones in software artifacts other than the source code [62], such as clones in higher level code structure [9], clones in the models of formal model based development [29], in UML domain models [117], UML sequence diagrams [86], in the graph based Matlab/Simulink models [100], duplication in requirement specification documents [62, 65], predicting clones

among domain entities [101], and even in Spreadsheets [46]. Definitions of clones must capture clones in all types of artifacts, not just source code. However, this paper focuses on the management of clones in the source code only.

B. Clone Management Activities

To manage clones, first they have to be identified. The result of clone detection forms clone documentation that records the location of code segments and their clone relationship. If the code base changes due to ongoing development, the changes and locations of the clones need to be tracked, and the documentation needs to be updated accordingly. The clone documentation may be analyzed to determine justification of clones or to find potential clones for removal. Visualization techniques can aid such analysis. Clones that are found to have justified reason to exist may be further documented and/or annotated. The candidates for refactoring can be scheduled for modification and/or removal. Upon the application of refactoring operations, a follow up verification may examine if the refactoring caused any change in program behaviour, and in accordance, may initiate roll-back and re-refactoring. Upon completion of refactoring the clone documentation needs to be updated for consistency.

The workflow for a typical clone management system may compose all these activities according to as summarized in Figure 4. In the following sections, we describe the state of the art in support for each of the clone management activities.

IV. Clone Detection

Over more than a decade of code clone research a number of techniques have been devised for the detection of code clones and many clone detection tools have been developed. In this section, we provide a brief summary of different clone detection techniques. More detailed descriptions of those techniques can be found in the corresponding papers and elsewhere [104, 107].

Tracking Clipboard Operations: This technique of clone detection is based on the assumption that programmers' copy-paste activities are the primary reason for the creation of code clones. So, the technique [28, 51, 131] simply tracks clipboard activities in the editor (inside IDEs such as Eclipse) when a programmer copies a code segment and reuses by pasting it. The copied and the pasted code segments are recorded as clone-pairs.

Metrics Comparison: Metrics based techniques [83, 89] are usually used to detect function clones. The techniques are based on the assumption that similar code fragments should yield very similar values for different software metrics (e.g., cyclomatic complexity, fan-in, fan-out). Typically, for the code segments a set of metrics are gathered into vectors. The differences in the vectors are calculated, where close vectors (e.g., measured by Euclidean distance) indicate that their corresponding code fragments are clones.

Texual Comparison: Text based techniques [32, 59] compare program text, typically line by line, with or without normalizing the text by renaming the identifiers, filtering out the comments and differences in the layout.

Token Based Comparison: In token based techniques [5, 33], the entire program is transformed into a stream of tokens (i.e., individual units/words of meaning) through lexical analysis. Then the token stream is scanned to find similar token subsequences, and the original code portions corresponding to those subsequences are reported as clones.

Syntax Comparison: Syntax comparison based techniques [12, 56, 95] are developed on the fact that similar code segments should also have similar syntactic structure. Thus, the program is parsed to produce a syntax tree, where similar subtrees indicate that their corresponding code segments are clones.

PDG Based Comparison: For a given program, a set of PDGs (Program Dependency Graphs) are produced based on the data and control dependencies among the statements of the program. The code segments corresponding to the isomorphic subgraphs are identified and reported as clones [47, 49, 78].

Hash Based Comparison: Recently, hash based techniques are getting attention for fast and scalable detection of near-miss clones where hash values are generated from source code and processed further for finding clones [116, 126].

Comparison of Low Level Form of Code: Instead of analyzing and comparing textual source code, the techniques analyze the lower level code (e.g., assembly code, Java Bytecode or .Net intermediate language) as obtained from the transformation by the compiler [2, 27, 74].

Other Techniques: Besides the aforementioned prominent techniques for clone detection, other techniques, such as formal methods [113], and combination of distinct techniques [105] were also approached. Tracing of abstract memory states during the execution of the program was also attempted to detect semantic clones [76].

As listed above there have been a great many state of the art clone detectors available. However, still little is known about the useful of the detected clones by different clone detectors. Furthermore, evaluation of the clone detectors is still an open challenge [104, 107] as we do not have reliable benchmarks except the tool comparison experiment of Bellon et al. [15] and the mutation based framework of Roy, Cordy and Svajlenko [106, 119]. The parameter settings of the clone detectors is another threat as shown by Wang et al. [129] as confounding configuration choice problem and conducted an extensive study considering six clone detectors to ameliorate the effects of the problem. Not to mention the issue of big data clone detection is a growing challenge for clone management and for many other related applications [118].

V. Integrated Clone Detection

There are many clone detection tools out there, each has its own strengths and weaknesses. However, for proactive clone management, the support for clone detection should be integrated with the development process. Therefore, we focus on those tools that integrate clone detection with an IDE or a version control system.

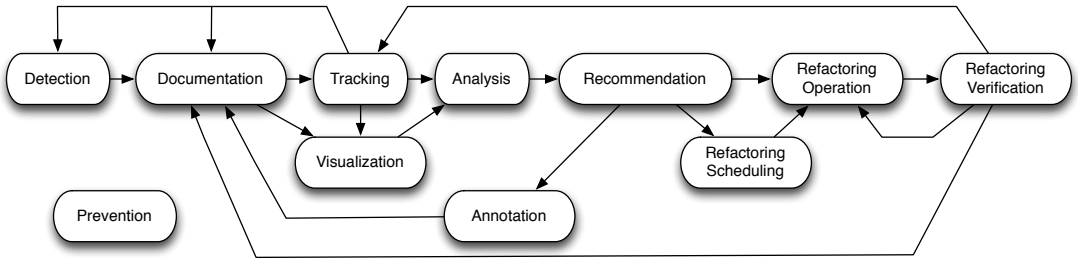

Fig. 4. Clone management workflow

Juergens et al. developed CloneDetective [64], an open source framework to facilitate implementation of customized clone detectors. The framework itself is built on the infrastructure of ConQAT[1], an integrated toolkit for software quality assessment. Currently, CloneDetective is an integral part of ConQAT, which applies a suffix-tree-based technique to detect *Type-1*, *Type-2*, and *Type-3* clones. However, beyond the detection of clones and visualization of the clone detection result, they offer no further support for clone management.

SimScan[2], which is a parser-based tool available as plugin to Eclipse, IDEA, or JBuilder, can detect *Type-1*, *Type-2*, and possibly a subset of *Type-3* clones. The potential of SimScan is also limited up to the detection code clones, not beyond that.

Giesecke [37] proposed a generic model for describing clones. The model allowed separation of concerns among the detection, description, and management of code clones. The objective was to ease the implementation of tools to support such activities. Based on the proposed model, they implemented DupMan, a framework [38] integrated with the Eclipse platform, and developed a prototype tool having SimScan as the back-end clone detector. The model and implementation is limited to the detection of clones and the representation of the clone information for persistence.

CloneBoard [28] and CPC [131] are Eclipse plugins similar to CloneScape that can detect and track clones based on clip-board (copy-paste) activities of programmers. Both CPC and CloneBoard support linked editing of clone pairs as described by Toomim et al. [125]. However, CPC was implemented as a framework to serve as a platform for future clone management technology, whereas, the focus of CloneScape was more on clone visualization and navigation, though their implementation remained incomplete. Hou et al. are developing a toolkit named CnP [51] for clone management, which also detects clones based on programmers' copy-paste activities. Indeed, the current implementation of CnP offers very limited support for clone management, which we address in Section X.

SHINOBI [73] is an add-on to the Microsoft Visual Studio 2005. For clone detection, it parses the source code, extracts sequences of pre-processed tokens and creates an index using suffix-tree based technique. SHINOBI internally uses CCFinderX's preprocessor, and thus it can detect *Type-1* and *Type-2* clones only, but not *Type-3* [143]. It was developed as a client(IDE)-server(CVS) application to mainly relocate the clone detection overhead from the client to a central server. It simply displays clones of a code fragment underneath the mouse cursor, no further support for clone management is offered. CodeRush[3] is a commercial add-in to the Microsoft Visual Studio for providing assistance in coding and refactoring. CodeRush recently introduced a new module DDC for the detection and consolidation of duplicated code.

Bahtiyar developed JClone [4] as a plugin for Eclipse for detecting code clones from Java projects. JClone applies an AST based technique to detect *Type-1* and *Type-2* clones only. It enables the user to trigger the detection of clones from one or more selected files or directories. It also offers a few visualizations (i.e., TreeMap and CloneGraph views) for aiding clone analysis to some extent, but no further support for clone management beyond the detection and visualization of clones.

Nguyen et al. developed JSync [95] as a plugin to the SVN version control system. Earlier prototypes of JSync appeared as Clever [98] and Cleman [96]. JSync detects clones based on similarities among the feature vectors computed over AST representation of the code fragments. JSync incorporates some useful features for clone management, which are discussed in Sections VII and X.

CPD[4] is a part of the Java source code analyzer, PMD. SDD [84] is a clone detection algorithm based on n-neighbour distance, index and inverted index. An implementation of SDD[5] is also freely available as a plugin to Eclipse. Simian[6] is another clone detector available as a plugin to Eclipse. Another Eclipse plugin, CloneDigger[7], applies an approach based on AST, suffix tree, and anti-unifcation for detecting clones in source code written in Java or Python. Tairas and Gray [121] also developed a suffix-tree based clone detector as a plugin for the Microsoft Phoenix framework. Despite the integration with IDEs all these tools offer no support for clone management except for the detection of only *Type-1* and *Type-2* clones [16].

[1]http://www.conqat.org/
[2]http://blue-edge.bg/download.html
[3]http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/
[4]http://pmd.sourceforge.net/cpd.html
[5]http://wiki.eclipse.org/index.php/Duplicated_code_detection_tool_(SDD)
[6]http://www.harukizaemon.com/simian
[7]http://clonedigger.sourceforge.net/download.html

Another Eclipse plugin, CloneDR[8], is an AST-based clone detector that can detect *Type-1* and *Type-2* clones. Besides clone detection, CloneDR offers support for clone removal as further discussed in Section X. CeDAR [122] can incorporate the results from different clone detection tools (e.g., CCFinder, CloneDR, DECKARD, Simian, or SimScan) and can display properties of the clones in an IDE. CeDAR offers no further support for clone management, except that those clone properties may be useful for clone analysis. Moreover, it may suffer from the limitations of the underlying clone detector used internally. Recently, Zibran and Roy [143] developed an Eclipse plugin to facilitate focused search for clones of a selected code fragment. They applied a suffix-tree-based k-difference hybrid approach to detect both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones. They are also extending their tool towards a versatile clone management tool [140].

While we see that there are a number of IDE-based clone detection tools available, there are only a few that in fact can deal with *Type-3* clones. Furthermore, as we will see in the following sections that there is still a marked lack for different clone management features in these IDE-based tools. Researchers possibly should first conduct user studies of what sort of features are needed for effective clone management and then start building tools that would help developers and maintenance engineers in dealing with different types of clones.

VI. Clone Documentation

Different clone detectors report the results of clone detectors in different formats such as XML, HTML, and plain text. There are variations in the reported information as well. Some clone detectors report clone pairs only, while some other tools report clones in terms of clone groups. Such variations make it difficult for data exchange between clone detectors, which also adds to the challenges in head-to-head empirical comparison of clone detectors. To minimize the differences in the presentation of clone information, Harder and Göde [44] recently proposed the *Rich Clone Format (RCF)*, an extensible schema based data format for storage, persistence, and exchange of clone data.

Duala-Ekoko and Robillard [30] proposed clone region descriptor (CRD) to describe clone regions within methods in a way that is independent of the exact text of the clone region or its absolute location in a file. However, such a scheme has a number of limitations. First, small changes in the code corresponding to the <*anchor*> (e.g., termination condition of loop, branching predicate of conditional statements) will invalidate the CRD. Second, the scheme is vulnerable to nesting levels, and thus a simple addition or removal of nesting level will invalidate the CRD. Third, the association of 'else' blocks with the closest 'if' block prevents the CRD scheme differentiating between the two types of blocks. Most importantly, the use of the CRD scheme did not save CloneTracker [30] from re-invoking the underlying clone detector to identify possible changes in the clones, though the computational expense of re-detection was indicated as one of the motivations behind the design of CRD.

The above discussion indicates that the line and column information, or the abstract level CRD based documentation of clone regions are more or less vulnerable to changes in the evolving code. To overcome such sensitivity to code change, marker based tagging support in IDEs like Eclipse can be used for clone documentation. Such tagging of clones can provide built-in support for accommodating changes in the source files [21]. Further investigation may be required to verify this possibility.

Capturing the location of clones reliably is necessary for tool comparisons and also for tracing clones over subsequent versions. If tools are to be integrated from different vendors, an agreed way to document clones is required. RCF is a step towards a common format [44], but it does not address all needs [72]. A common conceptual model for clone information is a major challenge because of competing requirements (e.g., it should be both generic and efficient) [42]. There has been some progress towards a unified model [72]. We expect real practical progress to happen, however, only if different research teams actually start to exchange data – and not just between two teams but among many teams. We do not see this happening at the moment except for exchanging benchmark data for clone detectors and for that use case, RCF seems to be sufficient.

VII. Clone Tracking

During the development of an evolving software system frequent changes take place in the code base. Such changes may introduce new code segments that might form new clones. Moreover, changes in source files may invalidate the clone regions necessitating corresponding updates in the recording of clone information. Such updates can be accomplished in two ways: Re-detection and incremental detection.

Re-detection: The detection of clones from the entire system may be invoked every time the code changes. This approach may incur too much overhead as the detection of code clones in a fairly large system can be computationally expensive. Hence, the approach might not be suitable for proactive clone management.

Incremental Detection: A better approach can be incremental detection, where only the source code in the modified portion of the code base is examined for any clones and the outcome is accumulated with the previously preserved clone detection results. Not many attempts were made towards incremental clone detection. The first attempt was made by Göde and Koschke [40]. They proposed a suffix tree based detector iClone for incremental detection of clones in subsequent versions of a given system. It detects *Type-1*, *Type-2*, and *Type-3* clones.

Hummel et al. [53] proposed an index-based incremental clone detection approach for *Type-1* and *Type-2* clones. Higo et al. [49] proposed an incremental one based on PDGs, where

[8]http://www.semdesigns.com/Products/Clone/

PDGs are generated from the analysis of control and data dependencies in the program code targeting even semantic clones. However, PDG based techniques are computationally expensive and they often report *non-contiguous* clones that may not be perceived as clones by a human evaluator [15]. The clone tracking approach of JSync [95, 97] appears to be computationally elegant. JSync preserves the clone-groups and N buckets obtained from the initial clone detection. Since JSync is implemented as a plugin to SVN, the change information of the source files are readily available, and based on that information JSync can determine the fragments modified, added to, or deleted from the code repository. JSync then removes from the clone-groups those fragments that were changed or deleted. Then the LSH technique is applied to the newly added and modified code fragments to place them in the buckets. Then the fragments in each bucket are compared pair-wise to update the clone-groups. Thus, the clone detection technique of JSync appears to be inherently incremental and consequently computationally efficient for tracking clones. We envision that other classes of techniques (cf., Section IV) will also have incremental variants as there is a clear need for scalable, fast and near-miss incremental detection techniques for efficient clone management.

VIII. Analysis of Clone Evolution

Software development and maintenance in practice follow a dynamic process. With the growth of the program source, code clones also experience evolution from version to version. Many studies have been conducted to date for understanding the overall evolution [8, 40, 112, 145], stability of cloned code [6, 45, 50, 81, 90, 91, 92], the relation of clone evolution with software faults [8, 39, 132, 133], and other characteristics of clone evolution. While such studies inform the characteristic and impact of code cloning, further in-depth analyses that investigate the change patterns in the evolution of individual clone fragments can suggest techniques for optimizing clone management including refactoring and removal. Because there is already a recent survey on clone evolution [99], we keep this section brief with specific focus on the evolution of individual clone fragments and their change patterns.

Kim et al. [77] first coined the term *"clone genealogy"*, which refers to a set of one of more lineage(s) originating from the same clone-group. A *clone lineage* is a sequence of clone-groups evolving over a series of versions of the software system. To map clones across subsequent versions of a program (i.e., extraction of clone genealogies) several approaches have been proposed in the literature. While most of these approaches [6, 8, 77, 108] focused on genealogies of *Type-1* and *Type-2* clone genealogies, gCad [110] is the only *Type-3* clone genealogy extractor to date released as a separate tool [111].

Studies [6, 16] on clone change patterns revealed that inconsistent changes in clones sometimes caused program faults. Moreover, late propagation is reported to have even more significant correlation with software defects and thus concluded to be "more risky than other clone genealogies" [8]. On the basis of clone genealogies, a number of studies [77, 108, 110, 112, 146] have been conducted to explore the change patterns and characteristics in the evolution of individual clone fragments. Some of the findings from those studies compliment one another, while some of the results derived from those studies appear to be contradictory. Therefore, more studies in larger scale are still necessary to confirm the agreeing observations and to shed light on the contradictory findings.

Studies on clone change patterns using a genealogy model can suffer from a number of issues. First, due to the threshold based similarity measure used in practice for *Type-3* clones, there remains an open question on the appropriate value for the threshold. Moreover, for *Type-3* clones, is it an appropriate practice to group *Type-3* clones into different disjoint classes? If not, the traditional notion of genealogy cannot apply to *Type-3* clones. Can we devise a more appropriate alternative? Second, a genealogy can be characterized as inconsistently changed if only a single clone over the entire length of the genealogy experiences even for very minor inconsistent changes. To draw a better picture, we may capture information such as, what portion of clones in clone groups change in how many versions, and how large the changes are. Finally, from the correlation between late propagation and software defects, can we really derive a causal relationship concluding that late propagation is riskier than other clone genealogies? Inconsistent changes are believed to often cause defects, and clones may disappear from a genealogy due to inconsistent changes. Later modifications, which could be even bug fixing activities, may cause changes in the disappeared clone to sync it to its original clone-group. In such a scenario, late propagation actually contributes in repairing the defect introduced from inconsistent changes. Thus, we believe, late propagation can really play a dual role, and more studies are necessary to distinguish them.

Visualization support can aid analysis of clone evolution, and thus different techniques and tools have been proposed for visualizing properties of clone evolution including the genealogy model. Adar and Kim [1] developed SoftGuess, a system for clone evolution exploration that supports three different views. The genealogy browser offers a simple visualization of clone evolution where nodes represent clones, arranged from left to right, and those that belong to the same class are arranged vertically in the same position. Thus, each column represents a version. A link between a pair of node reflects the predecessor and successor relationship during the evolution of the software. The encapsulation browser shows how clones within a clone group are distributed in different parts of a system and how they fit in the hierarchical organization of the software system by visualizing the containment relationship through a tree structure. Finally, the dependency graph describes how the nodes (package, class or method) within a version are evolved from other nodes and how they evolve in the next version. In addition, SoftGUESS also supports charting and filtering mechanisms based on Gython, an SQL-like query language. However, SoftGUESS lacks

an 'overview' feature and requires user interaction for data reduction through queries. Although a query is a powerful mechanism to identify important patterns of cloning, formulating queries could be difficult as this requires more cognitive effort from the developers.

Harder and Göde [44] developed a multi-perspective tool for clone evolution analysis, called CYCLONE. It offers five different views to analyze clone data stored in an RCF file, where RCF is a binary format to encode clone data including the evolutionary characteristics. The evolution view in CYCLONE visualizes clone genealogies using simple rectangles and circles to denote software entities. Each circle represents a clone fragment arranged in a set of rows where each row represents a particular version of the software. The clones that belong to the same clone class are packed within a rectangle. Finally, lines represent the evolution of a clone fragment. In addition, the view employs colors to distinguish types and the changes of the clones. Although the view highlights many important evolutionary characteristics, the volume of data produced by the genealogy extractor still limits its usefulness, thus calls for overview and filtering mechanisms. A similar visualization support is available in VisCad [3], with additional flexibility of metric-based filtering of genealogies.

While there have been a good number of studies (c.f., Section XI) on visualization of clones in a single version of a software system, we still need further studies to figure out useful techniques for visualizing clone evolution from management perspective. For example, what sort of visualizations are useful for clone management activities and what are their implications in the context of real world software development? We need to understand the claims and believes about code clones [18] including empirical evidence from developers' perspective [20, 146] and then need to design the visualization techniques appropriately. It is also important to understand developers' intent when designing such tools [19].

Recently, Saha et al. [109] presented an idea for clone evolution visualization using the popular scatter plot. In their proposed approach, scatter plots show the clone pairs associated within a pair of software unit (file, directory or package). Based on the type of clone genealogies they are associated with, clone pairs are rendered with different colours. Selecting a clone pair through user interactions (double clicking on a clone pair in the scatter plot) shows the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone classes in a particular version and then provide a way to call for genealogy browser to dig deeper. However, it does not provide overall characteristics of the genealogies. Moreover, due to the large number of clone pairs, selection and useful pattern identification in such a scatter plot can be difficult, which is why different variants of the traditional scatter plots appeared in the literature [24].

IX. Clone Annotation

The developers often deliberately create clones, for example, to enable independent evolution of similar implementations. During the clone management process, the developer may not want to refactor/remove those clones, and may want to mark those to indicate such decisions so that they will not have to encounter those same sets of clones over and over. Moreover, the decision needs to be documented and shared among different programmers, and there should be facilities for the developers to review those clones at a later time, in case they want to re-evaluate their management decision. To the best of our knowledge, such a feature is found only in JSync [95], which allows the developer to annotate pairs of clones for avoiding future encounters.

Although there are several ideas and implementations of clone-evolution visualization, there is not enough empirical assessment of these. We also believe that further progress can be achieved by studying existing work in information visualization.

X. Techniques for Reengineering/Refactoring of Clones

The investigations of opportunities for clone based reengineering and refactoring of clones for their removal have suggested techniques such as generics, design patterns, software refactoring patterns, and synchronized modifications of code clones.

Generics and Templates: Basit el al. [10] investigated the potential of generics in removing code clones. They carried out two case studies on the *Java Buffer Library* and the *C++ Standard Template Library (STL)*. The *Java Buffer Library* was found to have 68% redundant code, and using generics they were able to remove only 40% of them. Though, they performed little better for the *C++ STL*, they concluded that the constraints of language constructs limit the applicability of generics in clone removal. They further hypothesized that meta level parameterizations might perform better as they are relatively lesser restrictive than generics or templates.

The hypothesis on the potential of meta level parameterizations was addressed by Jarzabek and Li [55] in a later study. They also used the *Java Buffer Library* for their case study. They applied a generative programming technique using *XVCL (XML-based Variant Configuration Language)*[9] to represent similar (but not necessarily identical) classes and methods in generic and adaptable form. Using the technique they were able to eliminate 68% of the code from the original *Java Buffer Library*.

Consistent Renaming: Programmers often perform modifications after copy-pasting a code fragment. Such modification typically include renaming of identifiers according to the new context of the cloned code. IDEs like Eclipse provide necessary support for consistently renaming an identifier and all its references within scope. Jablonski and Hou developed CReN [54] as a plugin to Eclipse that can check for any inconsistencies in the renaming of identifiers within a code fragment and suggest modifications for making the renaming consistent.

[9] http://xvcl.comp.nus.edu.sg/

Since JSync [95] is developed as a plugin to the SVN version control system, it can exploit the change information between versions of Java source files to determine whether any changes occurred in cloned code regions.

Refactoring Patterns Fowler in his book [35] presented 72 patterns for refactoring source code in general for the removal of code smells. Over time, the number of refactoring patterns has increased to 93, and a *refactoring catalog*[10] is maintained that lists and describes them all. Among those general software refactoring patterns [35], *Extract method*, *Move method*, *Pull-up method*, *Extract superclass*, *Extract utility-class*, and *Rename refactor* patterns are found to be suitable for clone refactoring, as suggested by earlier research [48, 85, 114, 136, 141, 144]. Details about these refactoring patterns can be found in the *refactoring catalog* and elsewhere [35].

Besides these prominent refactoring patterns, other low level refactoring operations such as *identifier renaming*, *method parameter re-ordering*, *changes in type declarations*, *splitting of loops*, *substitution of conditionals*, *loops*, *algorithms*, and *relocation of methods or fields* may be necessary to produce generalized blocks of code from near-miss clones [141]. Kerievsky [75] proposed the *chained constructor* refactoring pattern[11], to eliminate duplicated code from the constructors of the same class [94]. Other refactoring patterns that can be found in the literature are some sort of variants or compositions of the aforementioned object-oriented refactoring (OOR) patterns. Other than the OOR patterns, Schulze et al. [115] proposed three aspect oriented patterns described as *extract feature into aspect, extract fragment into advice*, and *move method from class to interface*.

Type-3 clones remain a challenge for automated clone refactoring because they have difference that cannot be eliminated with a simple rename refactoring. Here, an additional step is required to abstract from the difference in a way that enables an extract-method refactoring. Anti-unification used to detect clones may help in refactoring, too, in certain situation. We expect progress for some *Type-3* clones at least. It is an interesting question how far we can get. A couple of recent studies [14, 82] also call for further studies on clone refactoring including the refactoring of task specific near-miss clones. Indeed, clone maintenance support could be increased by unifying clone detection and refactoring activities [123] and we need to focus more on such studies.

XI. Analysis and Identification of Potential Clones for Refactoring

For the purpose of finding and characterizing code clones suitable for refactoring, reengineering, or removal, in depth analysis of the various properties of the clones and their context is required. Clone visualization has been proven to be effective in aiding such analysis. Therefore, we first discuss the tools and techniques for code clone visualization, and then we present the findings from analysis of code clones in search for clone based reengineering opportunities.

A. Visualization of Distribution and Properties of Clones

Almost all the clone detection tools report clone information in the form of clone pairs and/or clone groups in a textual format where only the basic information about the clones such as the file name, line number, starting position, ending position of clones are provided. The returned clones also differ in several contexts such as types of clones, degree of similarity, granularity and size.

Moreover, there is a huge amount of clones in large systems. For example, *CCFinder* resulted 13,062 clone pairs for Apache httpd [67]. Because of the insufficient information on the returned clones, their various contexts, and their sheer number, the presentation of clones becomes difficult. For the proper use of the detected clones, especially for clone management, the aid of suitable visualization is crucial. In the following, we list some of the visualization approaches that have been proposed in the literature.

A major challenge in identifying useful cloning information is to handle the large volume of textual data returned by the clone detectors. To mitigate the problem, a number of visualization techniques, filtering mechanisms and support environments are proposed in the literature. Jiang et al. [58] categorized the proposed clone presentation techniques based on two dimensions. The first dimension refers to the level at which the entities are visualized (such as at the code segment level or file level or subsystem level). The second dimension refers to the type of clone relation addressed by the presentation, that is, whether clones are shown at the clone pair level or grouped into clone classes or super clones. A *super clone* is an aggregated representation of multiple clone groups between the same source entities (e.g., file).

Johnson [60] used the popular Hasse diagram to represent textual similarity between files. Later, he also proposed hyperlinked web pages to explore the files and clone classes [61]. Cordy et al. [25] used HTML for interactive presentation of clones where overview of the clone classes is presented in a web page with hyperlinks and users can browse the details of each clone class by clicking on those links. Although such representations offer quick navigation, they cannot reveal the high level cloning relations.

A set of polymetric views [103] were also proposed in the literature that permit encoding multiple code clone metrics in visual attributes. Among various visualizations, scatter plot is quite popular and capable of visualizing inter-system and intra-system cloning [24, 87]. However, the size of the scatter plot depends on the size of input rather than the amount of cloning. Thus, using a scatter plot for visualizing cloning relation of a large software system may become challenging due to the large size of the plot.

Moreover, in scatter plot, non-contiguous sections that contain the same clone cannot be grouped together. To overcome this limitation, Tairas et al. [124] proposed a graphical view of clones (also known as Visualizer view) that represents each

[10]Catalog of OO refactoring patterns: http://refactoring.com/catalog/

[11]Catalog of 27 refactoring patterns from J. Kerievsky's book: http://industriallogic.com/xp/refactoring/catalog.html

source file as a bar and clones within the files are represented with stripes. Clones belong to the same class are encoded with the same color.

Jiang et al. [57] extended the idea of cohesion and coupling to code clones and proposed a visualization that uses shape and color to encode the metric values. They also developed a framework [57] for large scale clone analysis and proposed another visualization, called a clone system hierarchical graph that shows the distribution of clones in different parts (with respect to the file-system hierarchy) of a system. Fukushima et al. [36] developed another visualization using graph drawing to identify diffused (scattered) clones. Here, nodes represent the clones. Those nodes that are located in the same file are connected with edges to form a clone set cluster. Nodes that connect different clone set clusters are called diffused clones (have cloning relation in different files implementing different functions).

`Gemini` [127] is an example of a clone support environment that uses CCFinder for clone detection and can visualize clone relationships using scatter plots and metric graphs. Kapser and Godfrey developed `CLICS` [70, 71], another tool for clone analysis. `CLICS` can categorize clones based on their previously developed clone taxonomy [69] and support query based filtering. Tairas et al. [124] developed an Eclipse plug-in that works with `CloneDR` as a clone detector and implements the visualizer view along with general information and detected clones list views.

`Clone Visualizer` [139] is an Eclipse plugin that works with `Clone Miner` to detect clones. In addition to supporting clone visualization through stacked bar charts and line graphs, it supports query based filtering. `CYCLONE`[12] [44] is another clone visualizer that supports single and multi-version program analysis and uses RCF (Rich Clone Format) [44] file as an input. A separate viewer application named `RCFVIEWER`[13] is also developed for the visualization of clone information stored in RCF. Recently, Xing et al. [134] proposed *CloneDifferentiator* that identifies contextual differences of clones and allows developers to formulate queries to distill candidate clones that are useful for a given refactoring task.

As can be noted, all the visualization techniques focus on visualization of clone pairs or clone groups with respect to their dispersion in the file-system hierarchy only. However, the cost-benefit analysis of code clone refactoring (Section XI-B) takes into account the distribution of clones in the inheritance hierarchy. Therefore, from the perspective of clone removal or refactoring, the visualization of the clones with respect to the inheritance hierarchy can offer useful insights, and future work in clone visualization should address this possibility.

For visualization of clone evolution, the proposed techniques for visualizing clones of one version of a system lacks empirical assessment mostly. We possibly need use-case specific visualizations with empirical support as of Live Scatterplots [23]. We thus expect to have more empirical user studies as the field matures. Furthermore, clone visualization from big data is also badly needed [34].

[12]http://softwareclones.org/cyclone.php

[13]http://www.softwareclones.org/

B. Cost-benefit Analysis and Scheduling of Refactoring

Not much research has been done towards cost-benefit analysis of code clone refactoring and their scheduling. Bouktif et al. [17] first proposed a simple effort model for the refactoring of clones in procedural code. Zibran and Roy [141, 142, 144] proposed a more comprehensive effort model for estimating clone refactoring efforts. They formulated scheduling of code clone refactoring as a constraint satisfaction optimization problem and applied constraint programming (CP) technique to compute an optimal solution of the problem.

Lee et al. [85] applied ordering messy GA (OmeGA) to schedule refactoring of code clones. Mondal et al. [93] proposed an automatic way of ranking clones for refactoring through mining association rules of the evolving clones. Juergens and Deissenboeck [63] described a detailed analytic cost model based on potential effects of clones on different maintenance activities. The existing models make several implicit and explicit assumptions and do not give concrete values for weights included in the formulae.

Overall, we know too little about the real costs incurred by clones and the risks and benefits of refactorings and other measures to compensate the negative effects of clones for a realistic cost model. We hardly know the factors influencing the costs. Only through a series of empirical field studies and experiments will we ever get closer to such a cost model. We remain skeptical as to whether we will ever get close enough given the many variables influencing the costs and gains of clones.

XII. Root causes for Code Duplication

Code clones do not occur in software systems by themselves. They are created. There are several factors that might force or influence the developers and/or maintenance engineers in cloning code in a system. In order to manage clones properly, we need to study the root causes for their creation. In the following we list some of the potential root causes.

Development Strategy Clones can be introduced in software systems due to the different reuse and programming approaches. Reusing code, logic, design and/or an entire system (as in product lines [31]) are the prime reasons of code duplication. Reusing existing code by copying and pasting (with or without minor modifications) is the simplest form of reuse mechanism in the development process which results in code duplication. It is a fast way of reusing reliable semantic and syntactic constructs.

The term *Forking* is used by Kapser and Godfrey [68] to mean the reuse of similar solutions with the hope that they will diverge significantly with the evolution of the system. For example, when creating a driver for a hardware family, a similar hardware family may already have a driver, and thus can be reused with slight modifications. Similarly, clones can be introduced when porting software to new platforms and

functionality and logic can be reused if there is already a similar solution available.

Maintenance Benefits Clones are also introduced in the systems to obtain several maintenance benefits. One of primary factors could be reducing risk in developing new code. Cordy [22] reports that clones do frequently occur in financial software as there are frequent updates/enhancements of the existing system to support similar kinds of new functionalities. Financial products do not change that much from the existing one, especially within the same financial institutions. The developer is often asked to reuse the existing code by copying and adapting to the new product requirements because of the high risk (monetary consequences of software errors can run into the millions in a single day) of software errors in new fragments and because existing code is already well tested (70% of the software effort in the financial domain is spent on testing). Introduction of new bugs can be avoided in critical system functionality by keeping the critical piece of code untouched [41]. For keeping the software architecture clean and understandable sometimes clones are intentionally introduced to the system [68].

Overcoming Underlying Limitations Clones can be introduced due to limitations of the programming language, especially when the language in question does not have sufficient abstraction mechanisms such as inheritance, generic types (called templates in C++) or parameter passing (missing from, e.g., assembly language and COBOL) and consequently, the developers are required to repeatedly implement these as idioms. Such repeating activities may create possibly small and potentially frequent clones [11].

There are also several limitations associated with the programmers for which clones are introduced in the system. For example, it is generally difficult to understand a large software system. This forces the developers to use the example-oriented programming by adapting existing code developed already. Furthermore, often developers are assigned short time frames in completing tasks. Due to such time limits, developers look for an easy way of solving the problems at hand and consequently look for similar existing solutions and consequently clones are introduced in software. Sometimes the productivity of a developer is measured by the number of lines he/she produces per hour. In such circumstances, the developer's focus is to increase the number of lines of the system and hence tries to reuse the same code again and again by copying and pasting with adaptations instead of following a proper development strategy. Sometimes the developer is not familiar with the problem domain at hand and hence looks for existing solutions of similar problems.

Cloning By Accident Clones may be introduced into software even by accidents. The use of a particular API normally needs a series of function calls and/or other ordered sequences of commands. For example, when creating a button using the Java SWING API, a series of commands is to create the button, add it to a container, and assign the action listeners. Similar orderings are common with libraries as well [68]. Thus, the uses of similar APIs or libraries may introduce clones. Coincidentally implementing the same logic by different developers may also cause cloning. Programmers may unintentionally repeat a common solution for similar kinds of problems using the common solution pattern of his/her memory to such similar problems. Therefore, several clones may unknowingly be created in the software systems.

As can be seen from the above discussion, we need to dig deeper into each of the root causes so that we can either avoid clones or can keep track of the clones during development making clone management easier in the maintenance as also noted by Zhang et al. [137]. We can also think of better programming languages design keeping more abstraction mechanisms for different types of clones at the fingers end of the developers.

XIII. Clone Management Strategies

For dealing with code clones, Mayrand et al. [88] proposed two concrete activities namely "problem mining" and "preventive control", which were further supported by a later study of Lague et al. [83]. Giesecke [37] categorized them into compensatory and preventive clone management, respectively. Giesecke [37] suggested that all clone management activities can be associated with one or more of the three categories: corrective, preventive, and compensatory management.

Corrective clone management aims for *removal* of existing clones from the system. The objective of *Preventive clone management* is to prevent creation of new clones in the system. *Compensatory clone management* deals with applying techniques (such as annotation, documentation) for avoiding the negative impacts of clones that are not removed from the system for some valid reasons. In practical settings, avoiding clones may be impossible at times, and the expectation of a clone-free system can be unrealistic. Thus, *preventive* clone management actually refers to *proactive* management [51, 52] that aims to deal with the clones during their creation or soon after they are introduced. An opposite strategy, *retroactive clone management* [21] adopts the *post-mortem approach* [143], where clone management activities initiate after the development process is complete up to a milestone.

Clone management in legacy systems can be the most appropriate for the *post-mortem* strategy. Indeed, prevention is better than cure. Therefore, *proactive* clone management is preferable to *post-mortem* approach. While, ideally, all clones should be managed proactively, in practical settings, proactive treatment for all clones may not be feasible or possible. Therefore, a versatile clone management system should focus on support for proactive management, while at the same time, should also facilitate retroactive clone management [21]. Recently, Zhang et al. [138] proposed, *CCEvents* that provides timely notifications about relevant code cloning events for different stakeholders through continuous monitoring of code repositories. This is one of the first studies on contextual and on-demand clone management that clearly shows we need further studies on clone management as well.

XIV. Design Space for a Clone Management System

Most clone detectors [47, 56, 66, 105] are implemented as stand-alone tools separate from IDEs (Integrated Development Environments) and typically search for all clones in a given code base. While clone detection from such tools can help clone management in a post-mortem approach, researchers and practitioners [37, 43, 51, 52, 83, 95, 140, 143] believe that clone management activities should be integrated with the development process to enable proactive management.

Hou et al. [52], during the on-going development of their clone management tool `CnP` [51], explored the design space towards tool support for clone management. However, their work was tightly coupled with the clone detection technique based on the programmers' copy-paste operations. Thus, their findings are limited in scope to the management of copy-pasted code, and most of the findings are not applicable to clone management based on similarity based clone detection.

A. Architectural Alternatives of Integration

We identify three major alternatives and some sub-alternatives in the design space for a versatile clone management tool. These alternatives are inspired by our experience and the different clone management scenarios reported by Giesecke [37].

Architectural Centrality: The need for the integration of clone management activities with the development process suggests that the IDEs should include features to support clone management activities during the actual development phase. While a programmer typically works inside an IDE running on her individual workstation, for fairly large projects, especially in industrial settings, a team of developers collaboratively works on a shared code base kept in a version control system set up on a server. Hence, the supports for clone management activities can be implemented as features augmenting the local IDEs, or the functionalities can be implemented at a central repository.

Decentralized Architecture: The clone management functionalities, when augmenting the features in local IDEs, can enable the individual programmers to exploit the benefit of clone management. In the decentralized scenario different developers can use different tools, and some programmers can get the flexibility to completely or partially disregard clone management at their respective situations. Apparently, such a decentralized implementation may completely disregard the existence of a central server, and enforces proactive clone management before check-in to the shared repository. However, this necessitates additional requirements for establishing means for communication between distributed developers, as well as combining and synchronizing clone information across all the developers.

Centralized Architecture: The centralized architecture inherently aims to support clone management in a distributed development process. The functionalities can be implemented as a client-server application on top of central version control systems. Such a centralized clone management system may require greater effort and offer less flexibility than a decentralized implementation [37]. Indeed, a client-server implementation cannot support those individual programmers who work alone on their stand-alone local machines [143]. But, the centralized architecture may facilitate the integration of clone detection features with the continuous or periodic (e.g., diurnal) build process [135].

We currently do not know what strategy works best under which circumstances. Future research should compare the different ways of integrating clone management in the development process.

B. Triggering Actors in Clone Management

A clone management activity may be initiated by the developer or by the system in response to certain events.

Human Triggered Initiative: A developer, after writing or modifying a piece of code, may invoke the search for clones in the system, upon finding the clones, she may analyze and decide how to deal with them. In such an ad-hoc triggering scenario, the developer, at times, may forget to perform the necessary clone management. An instance of clone management activity may also be periodically scheduled in advance as part of a larger plan of process activities, and clone management activities can be carried out following the post-mortem approach on the current status of the code base.

System Triggered Initiative: The development environment can trigger clone management activities in response to certain events, such as saving changes in the code, or check-in of modified code to the central repository having the clone detection capability integrated with the build process. Such events may notify and suggest the developer to perform the required clone management operations. However, care must be taken so that those auto-generated notifications and suggestions do not irritate the developer or hinder her normal flow of work.

Scope of Clone Management Activity: An instance of clone management activity may be *clone-focused* or *system-focused*. A clone-focused activity deals with a narrow set of clones of a particular code segment of interest. On the contrary, a system-focused clone management activity aims to deal with a broad collection of clones in the entire code base, or particular portions of the system.

We need to further investigate for which kind of events clone management should be triggered by whom. For changes in clones – in particular inconsistent ones – likely a system should notify a developer. The challenge for all actions triggered by a system must be to avoid false alarms. Otherwise developers will soon give up using a clone management tool. For general quality assurance, a quality manager might observe trends in clones and take initiatives when she sees a increase of redundancy. The challenge for human triggered actions is to provide accurate data on demand and to find significant indicators of problems.

C. Scope and Point in Time of Clone Management

Clones are not restricted to source code. Finding clones in earlier artifacts may avoid source code clones.

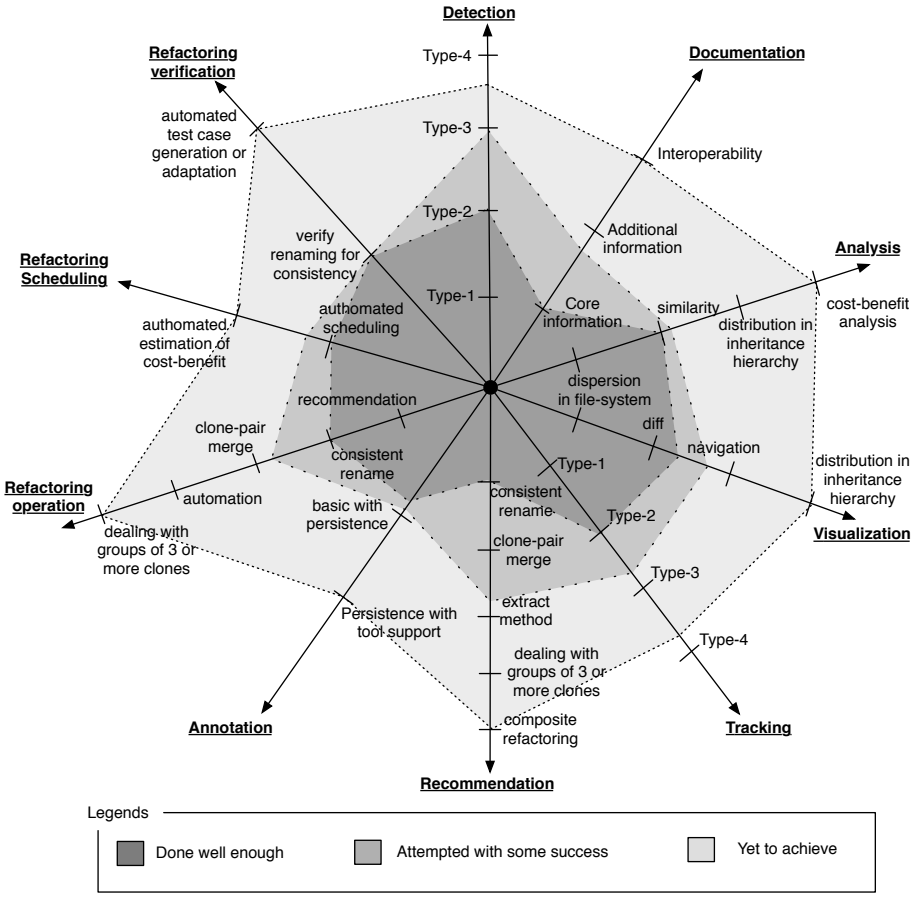

Fig. 5. Achievements and scopes along different dimensions of clone management activities

Clones in requirements documents may lead to duplicate implementation of very similar features if, for example, similar use cases are given to different development teams to implement. In turn, this may result in semantic code clones, or even code segments with very similar structure. Therefore, the detection of clones in requirements specification artifacts could help avoiding clones in code, or to identify semantic clones, which in turn could help to differentiate features from sub-features.

Similarly, clones in design models, especially in model-driven development, may also lead to clones in the source code. Thus, the upfront detection of clones in design models might help to reconsider architectural choices, and result in a leaner, more abstract and essential design resulting in fewer code clones.

For these reasons, clone management must consider all types of software artifacts and should be part of early stages in software development. There are initial studies in detecting clones in other types of artifacts such as in requirements documents [62, 65], in models [29, 117], in sequence diagrams [86] or in Spreadsheets [46], which need to explore and deepen further. We envision clone management tools to broaden from source code to other artifacts and as a consequence a good chance to avoid source-code clones.

XV. Industrial Adoption of Clone Management

Despite the active research on software clones and their impact on the development and maintenance of software system, management of code clones is still far from wide industrial adoption. A reason to this could be the unavailability of integrated tool support for versatile clone management. Or maybe industry is just not aware of the problem. Maybe clones are even not a real problem in the first place because the advantages outweigh the disadvantages.

What we as researchers need to show first is sufficient empirical evidence of real problems caused by clones. We have made good progress in recent years here. Then we need to provide usable working solutions. We need to demonstrate their benefits in real case studies. Because benefits are expected to show up only in the long run, we need long-term studies in realistic industrial settings. Such long-term industrial studies are difficult to conduct, however.

Despite these difficulties, we see signs that clone management is gathering momentum in industry. There are several clone detectors available as Eclipse plug-ins and only recently Microsoft introduced a clone management feature in Microsoft Visual Studio [26, 130]. There are several other industrial attempts as well [128, 135] including a recent Dagstuhl seminar on the topic [13].

XVI. Conclusion

In Figure 5, we summarize the state of the art along the different dimensions of code clone management and scopes for further improvements. Although software clone research matured over the last decade, the majority of the work focused on the detection and analysis of code clones. Compared to those, clone management has earned recent interest due to its practical importance. Notably several surveys [80, 99, 102, 104, 107] appeared in the literature, none of which focused on clone management, and thus a survey on clone management was a timely necessity. This paper presents a comprehensive survey on clone management and pin-points research achievements and scopes for further work towards a versatile clone management system.

At the fundamental level, the vagueness in the definition of clones at times causes difficulties in formalization, generalization, creation of benchmark data, as well as comparison of techniques and tools. A set of task oriented definitions or taxonomies can address these issues. Most of the integrated tools have limitations in detecting *Type-3* clones, and the detection of *Type-4* clones has still remained an open problem. Moreover, most of the research on software clones so far emphasized clone analysis at different levels of granularity. A variety of techniques for the visualization of clones and the evolution has been proposed. Surprisingly, while clone analysis points to the importance of considering inheritance hierarchy for extracting clone reengineering candidates, there is still not enough visualization support to analyze clones with respect to their existence in the inheritance hierarchy.

Research on clone management beyond detection has mostly been limited to devising techniques to identify clones. While detection is a necessity for clone management and many improvements have been achieved here, filtering and ranking relevant findings is still a major challenge. It is not yet clear what constitutes a bad clone that requires treatment. Neither is it sufficiently known what kind of treatment (refactoring or other types of compensation) works best under which circumstances. For the bad clones, we need to conduct root-cause analysis to better understand why they came into existence and how they could be avoided.

The state of the art demands more research in semi-automated tool support for clone refactoring and cost-benefit analysis of clone removal/refactoring. For integrated clone management, `JSync` [95] offers a relatively wide set of features compared to others. But, we see that the state of the art is still far from *integrated* tool support, and more is to be done towards a versatile clone management system. Perhaps, due to the unavailability of such tools, there is not much developer-centric ethnographic studies on the patterns of clone management in practice, as well as on the usability and effectiveness of tool support. This survey exposes such potential avenues for further research to create a better impact in the community.

Acknowledgement: We would like to thank our anonymous reviewers for their useful suggestions and critiques.

References

[1] E. Adar and M. Kim. SoftGUESS: Visualization and exploration of code clones in context. In *ICSE*, pages 762 –766, 2007.

[2] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *WCRE*, pages 405–414, 2012.

[3] M. Asaduzzaman, C. K. Roy, and K. A. Schneider. VisCad: Flexible code clone analysis support for NiCad. In *IWSC*, pages 77–78, 2011.

[4] M. Bahtiyar. JClone: Syntax tree based clone detection for Java. Master's thesis, Linnaeus University, 2010.

[5] B. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86 –95, 1995.

[6] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM*, pages 24–33, 2007.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pages 292 –303, 1999.

[8] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Soft.: Evol. and Proc.*, pages –, 2013 (in press). doi: 10.1002/smr.1597.

[9] H. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30:156–165, 2005.

[10] H. Basit, D. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *SEKE*, pages 109–114, 2005.

[11] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *ICSE*, pages 451–459, 2005.

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

[13] I. D. Baxter, M. Conradt, J. R. Cordy, and R. Koschke. Software clone management towards industrial application (dagstuhl seminar 12071). *Dagstuhl Reports*, 2(2):21–57, 2012.

[14] S. Bazrafshan and R. Koschke. An empirical study of clone removals. In *ICSM*, pages 50–59, 2013.

[15] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Softw. Engg.*, 33(9):577–591, 2007.

[16] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Science of Comp. Prog.*, 77(6):760 – 776, 2012.

[17] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *GECCO*, pages 1885–1892, 2006.

[18] D. Chatterji, J. C. Carver, and N. A. Kraft. Claims and beliefs about code clones: Do we agree as a community? a survey. In *IWSC*, pages 15–21, 2012.

[19] D. Chatterji, J. C. Carver, and N. A. Kraft. Cloning: The need to understand developer intent. In *IWSC*, pages 14–15, 2013.

[20] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder. Effects of cloned code on software maintainability: A replicated developer study. In *WCRE*, pages 112–121, 2013.

[21] A. Chiu and D. Hirtle. Beyond clone detection. CS846 Course Project Report, University of Waterloo, 2007.

[22] J. R. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *IWPC*, pages 196–206, 2003.

[23] J. R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. In *ICPC*, pages 151–160, 2011.

[24] J. R. Cordy. Live scatterplots. In *IWSC*, pages 79–80, 2011.

[25] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON*, pages 1–12, 2004.

[26] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: tuning code clones at hands of engineers in practice. In *ACSAC*, pages 369–378, 2012.

[27] I. Davis and M. Godfrey. Clone detection by exploiting assembler. In *IWSC*, pages 77–78. ACM, 2010.

[28] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, pages 169–178, 2009.

[29] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE*, pages 603–612. ACM, 2008.

[30] E. Duala-Ekoko and M. Robillard. Clone region descriptors: Repre-

senting and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20:3:1–3:31, 2010.
[31] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, pages 25–34, 2013.
[32] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109 –118, 1999.
[33] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13:601–643, 2008.
[34] C. Forbes, I. Keivanloo, and J. Rilling. Doppel-Code: A clone visualization tool for prioritizing global and local clone impacts. In *COMPSAC*, pages 366–367, 2012.
[35] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
[36] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida. Code clone graph metrics for detecting diffused code clones. In *APSEC*, pages 373 –380, 2009.
[37] S. Giesecke. Generic modelling of code clones. In *DRSS*, pages 1–23, 2007.
[38] S. Giesecke. Dupman - Eclipse duplication management framework, last access: Dec 2011. URL http://sourceforge.net/projects/dupman/.
[39] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311–320, 2011.
[40] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *J. of Soft.: Evol. and Proc.*, 25(2):165–192, 2013.
[41] T. L Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Tran. on Soft. Engg.*, 26(7):653–661, 2000.
[42] J. Harder. The limits of clone model standardization. In *IWSC*, pages 10–11, 2013.
[43] J. Harder and N. Göde. Quo vadis, clone management? In *IWSC*, pages 85–86, 2010.
[44] J. Harder and N. Göde. Efficiently handling clone data: RCF and cyclone. In *IWSC*, pages 81–82. ACM, 2011.
[45] J. Harder and N. Göde. Cloned code: stable code. *Journal of Soft.: Evol. and Proc.*, 25(10):1063–1088, 2013.
[46] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen. Data clone detection and visualization in Spreadsheets. In *ICSE*, pages 292–301, 2013.
[47] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *WCRE*, pages 315 –316, 2009.
[48] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. *PROFES*, pages 220–233, 2004.
[49] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *WCRE*, pages 3 –12, 2011.
[50] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *IWPSE-EVOL*, pages 73–82, 2010.
[51] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pages 238–242, 2009.
[52] D. Hou, F. Jacob, and P. Jablonski. Exploring the design space of proactive tool support for copy-and-paste programming. In *CASCON*, pages 188–202, 2009.
[53] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1 –9, 2010.
[54] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX*, pages 16–20, 2007.
[55] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software: Evolution and Process*, 18(4):267–292, 2006.
[56] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.
[57] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *SCAM*, pages 203 – 212, 2007.
[58] Z. Jiang, A. Hassan, and R. Holt. Visualizing clone cohesion and coupling. In *APSEC*, pages 467–476, 2006.
[59] J. Johnson. Substring matching for clone detection and change tracking. In *ICSM*, pages 120 –126, 1994.
[60] J. Johnson. Visualizing textual redundancy in legacy source. In *CASCON*, pages 32–41. IBM Press, 1994.
[61] J. Johnson. Navigating the textual redundancy web in legacy source. In *CASCON*, pages 16–25. IBM Press, 1996.
[62] E. Juergens. Research in cloning beyond code: a first roadmap. In *IWSC*, pages 67–68, 2011.
[63] E. Juergens and F. Deissenboeck. How much is a clone? In *SQM*, 2010.
[64] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.
[65] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE*, pages 79 –88, 2010.
[66] 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.*, 28(7):654–670, 2002.
[67] C. Kapser. *Toward an Understanding of Software Code Cloning as a Development Practice*. PhD thesis, University of Waterloo, 2009.
[68] C. Kapser and M. Godfrey. Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008.
[69] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE*, pages 85–94, 2004.
[70] C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM*, pages 305–314, 2005.
[71] C. Kapser and M. W. Godfrey. Supporting the analysis of clones in software systems: A case study. *J. Softw. Maint. Evol.*, 18:61–82, 2006.
[72] C. Kapser, J. Harder, and I. Baxter. A common conceptual model for clone detection results. In *IWSC*, pages 72–73, 2012.
[73] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A tool for automatic code clone detection in the IDE. In *WCRE*, pages 313–314, 2009.
[74] I. Keivanloo, C. K. Roy, and J. Rilling. SeByte: Scalable clone and similarity search for bytecode. *Science of Comp. Prog.*, pages –, 2013 (in press). doi: http://dx.doi.org/10.1016/j.scico.2013.10.006.
[75] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.
[76] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *ICSE*, pages 301–310, 2011.
[77] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *FSE*, pages 187–196, 2005.
[78] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.
[79] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE*, pages 44 –54, 1997.
[80] R. Koschke. Survey of research on software clones. In *DRSS*, pages 1–24, 2006.
[81] J. Krinke. Is cloned code more stable than non-cloned code? *SCAM*, 0:57–66, 2008.
[82] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *CSMR-18/WCRE-21 Software Evolution Week*, page 10, 2014 (to appear).
[83] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321, 1997.
[84] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *OOPSLA*, pages 140–141, 2005.
[85] S. Lee, G. Bae, H. Chae, D. Bae, and Y. Kwon. Automated scheduling for clone-based refactoring using a competent ga. *Softw. Pract. Exper.*, 41(5):521–550, 2010.
[86] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC*, pages 269 –276, 2006.
[87] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE*, pages 106–115, 2007.
[88] J. Mayrand, B. Lague, and J. Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *WESS*, 1996.
[89] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244 –253, 1996.

[90] M. Mondal, C. K. Roy, M. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *ACM-SAC*, pages 1227–1234, 2012.

[91] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *ACM Applied Comp. Review*, 12(3):20–36, 2012.

[92] 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. *Sci. of Comp. Prog.*, pages –, 2013 (in press).

[93] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *CSMR-18/WCRE-21 Software Evolution Week*, page 10, 2014 (to appear).

[94] S. Nasehi, G. Sotudeh, and M. Gomrokchi. Source code enhancement using reduction of duplicated code. In *IASTED*, pages 192–197, 2007.

[95] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Trans. on Softw. Engg.*, 1(1): 1–19, 2011.

[96] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE*, pages 451–454, 2008.

[97] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *ICSM*, pages 491–494, 2009.

[98] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.

[99] J. Pate, R. Tairas, and N. Kraft. Clone evolution: a systematic review. *Journal of Soft.: Evol. and Proc.*, pages 1–23, 2011.

[100] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*, pages 276–286, 2009.

[101] M. S. Rahman, A. Aryani, C. K. Roy, and F. Perin. On the relationships between domain-based coupling and code clones: an exploratory study. In *ICSE*, pages 1265–1268, 2013.

[102] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Infor. and Soft. Tech.*, 55(7):1165 – 1199, 2013.

[103] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[104] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Tech Report TR 2007-541, Queens University, 2007.

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

[106] C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *ICSTW*, pages 157–166, 2009.

[107] 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.*, 74:470–495, 2009.

[108] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.

[109] R. K. Saha, C. K. Roy, and K. A. Schneider. Visualizing the evolution of code clones. In *IWSC*, pages 71–72. ACM, 2011.

[110] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *ICSM*, pages 293 –302, 2011.

[111] R. K. Saha, C. K. Roy, and K. A. Schneider. gcad: A near-miss clone genealogy extractor to support clone evolution analysis. In *ICSM*, pages 488–491, 2013.

[112] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *MSR*, pages 139–148, 2013.

[113] A. Santone. Clone detection through process algebras and Java bytecode. In *IWSC*, pages 73–74. ACM, 2011.

[114] S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *WSR*, pages 1–2, 2009.

[115] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *WRT*, pages 6:1–6:4, 2008.

[116] N. Schwarz, M. Lungu, and R. Robbes. On how often code is cloned across repositories. In *ICSE-NIER*, pages 1289–1292, 2012.

[117] H. Störrle. Towards clone detection in UML domain models. In *ECSA*, pages 285–293, 2010.

[118] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *IWSC*, pages 16–22, 2013.

[119] J. Svajlenko, C. K. Roy, and J. R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *IWSC*, pages 8–9, 2013.

[120] R. Tairas. Code clones literature, (last access: Dec. 2013). URL http://students.cis.uab.edu/tairasr/clones/literature/.

[121] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE*, pages 679–684, 2006.

[122] R. Tairas and J. Gray. Get to know your clones with CeDAR. In *OOPSLA*, pages 817–818, 2009.

[123] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Info. & Soft. Tech.*, 54(12):1297–1307, 2012.

[124] R. Tairas, J. Gray, and I. Baxter. Visualizing clone detection results. In *ASE*, pages 549–550. ACM, 2007.

[125] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *VLHCC*, pages 173–180, 2004.

[126] M. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *WCRE*, pages 13 –22, 2011.

[127] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *METRICS*, pages 67–76. IEEE Computer Society Press, 2002.

[128] R. D. Venkatasubramanyam, S. Gupta, and H. K. Singh. Prioritizing code clone detection results for clone management. In *IWSC*, pages 30–36, 2013.

[129] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *ESEC/SIGSOFT FSE*, pages 455–465, 2013.

[130] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *ASE*, pages 170–179, 2012.

[131] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, Germany, 2008.

[132] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *MSR*, pages 149–158, 2013.

[133] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo. An empirical study on the fault-proneness of clone migration in clone genealogies. In *CSMR-18/WCRE-21 Software Evolution Week*, page 10, 2014 (to appear).

[134] Z. Xing, Y. Xue, and S. Jarzabek. Distilling useful clones by contextual differencing. In *WCRE*, pages 102–111, 2013.

[135] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T.i Sano. Applying clone change notification system into an industrial development process. In *ICPC*, pages 199–206, 2013.

[136] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, pages 16–25, 2005.

[137] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *ICSM*, pages 285–294, 2012.

[138] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao. Towards contextual and on-demand code clone management by continuous monitoring. In *ASE*, pages 497–507, 2013.

[139] Y. Zhang, H. Basit, S. Jarzabek, D Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *ICSM*, pages 376 –385, 2008.

[140] M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *IWSC*, pages 75–76, 2011.

[141] M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011.

[142] M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *ICPC*, pages 266 – 269, 2011.

[143] M. F. Zibran and C. K. Roy. IDE-based real-time focused search for near-miss clones. In *ACM-SAC*, pages 1235–1242, 2012.

[144] M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of prioritized code clone refactoring. *IET Software*, 7(3), 2013.

[145] 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*, pages 295–304, 2011.

[146] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Evaluating the conventional wisdom in clone removal: a genealogy-based empirical study. In *SAC*, pages 1123–1130, 2013.