

Software clone detection: A systematic review

Dhavleesh Rattan^{a,*}, Rajesh Bhatia^{b,1}, Maninder Singh^{c,2}

^a Department of Computer Science and Engineering and Information and Technology, Baba Banda Singh Bahadur Engineering College, Fatehgarh Sahib 140 407, Punjab, India

^b Department of Computer Science and Engineering, Deenbandhu Chhotu Ram University of Science and Technology, Murthal (Sonapat) 131 039, Haryana, India

^c Computer Science and Engineering Department, Thapar University, Patiala 147 004, Punjab, India

ARTICLE INFO

Article history:

Received 21 June 2011

Received in revised form 29 December 2012

Accepted 21 January 2013

Available online 14 February 2013

Keywords:

Software clone

Clone detection

Systematic literature review

Semantic clones

Model based clone

ABSTRACT

Context: Reusing software by means of copy and paste is a frequent activity in software development. The duplicated code is known as a *software clone* and the activity is known as *code cloning*. Software clones may lead to bug propagation and serious maintenance problems.

Objective: This study reports an extensive systematic literature review of software clones in general and software clone detection in particular.

Method: We used the standard systematic literature review method based on a comprehensive set of 213 articles from a total of 2039 articles published in 11 leading journals and 37 premier conferences and workshops.

Results: Existing literature about software clones is classified broadly into different categories. The importance of semantic clone detection and model based clone detection led to different classifications. Empirical evaluation of clone detection tools/techniques is presented. Clone management, its benefits and cross cutting nature is reported. Number of studies pertaining to nine different types of clones is reported. Thirteen intermediate representations and 24 match detection techniques are reported.

Conclusion: We call for an increased awareness of the potential benefits of software clone management, and identify the need to develop semantic and model clone detection techniques. Recommendations are given for future research.

© 2013 Elsevier B.V. All rights reserved.

Contents

1. Introduction & motivation	1166
1.1. Motivation for work	1166
2. Background	1167
2.1. Software clones	1167
2.2. Types of clones	1167
2.3. Why clones	1167
2.4. Advantages of clones	1167
2.5. Disadvantages of clones	1167
3. Review method	1168
3.1. Planning the review	1168
3.2. Research questions	1168
3.3. Sources of information	1168
3.3.1. Additional sources	1168
3.4. Search criteria	1169
3.5. Inclusion and exclusion criteria	1170
3.6. Quality assessment	1170
3.7. Data extraction	1170

* Corresponding author. Tel.: +91 9814837334; fax: +91 1763232113.

E-mail addresses: dhavleesh@rediffmail.com (D. Rattan), rbhatia@patiala@gmail.com (R. Bhatia), msingh@thapar.edu (M. Singh).

¹ Tel.: +91 9467948996; fax: +91 130 2484004.

² Tel.: +91 9815608309; fax: +91 175 2364498.

4.	Results.	1170
4.1.	Current status of clone detection.	1170
4.1.1.	Intermediate source representations and match detection techniques.	1170
4.1.2.	Clone detection tools.	1173
4.1.2.1.	Text based clone detection techniques.	1173
4.1.2.2.	Token based clone detection techniques.	1174
4.1.2.3.	Tree based clone detection techniques.	1174
4.1.2.4.	Graph based clone detection techniques.	1175
4.1.2.5.	Metrics based clone detection techniques.	1175
4.1.2.6.	Hybrid clone detection techniques.	1175
4.1.3.	Comparison and evaluation of clone detection tools and techniques.	1176
4.2.	Status of research in semantic and model clone detection techniques.	1179
4.2.1.	Semantic clone detection.	1179
4.2.2.	Model based clone detection.	1180
4.3.	Key sub areas.	1181
4.3.1.	Code clone evolution.	1182
4.3.2.	Code clone analysis.	1182
4.3.3.	Impact of software clones on software quality.	1184
4.3.4.	Clone detection in websites.	1186
4.3.5.	Cloning in related areas.	1186
4.3.6.	Software clone detection in aspect oriented programming/cross-cutting concerns.	1186
4.4.	Current status of clone management.	1187
4.4.1.	Benefits of clone management.	1187
4.4.2.	Clone management – a cross cutting and an umbrella activity.	1187
4.4.2.1.	Clone visualization.	1187
4.4.3.	Clone management: a systematic map.	1188
4.5.	Subject systems.	1188
5.	Discussion.	1190
5.1.	Key sub areas.	1190
5.2.	Clone management – a cross cutting topic.	1190
5.3.	Implications for research and practice.	1191
5.4.	Limitations of this review.	1191
6.	Conclusions and future work.	1191
	Acknowledgements.	1192
	Appendix A. A quality assessment forms.	1192
A.1.	Screening question.	1192
A.2.	Screening question.	1192
A.3.	Detailed questions.	1192
A.4.	Detailed questions.	1193
	Appendix B. Data items extracted from all papers.	1193
	Appendix C. Journals/conferences reporting most clone related research.	1193
	Appendix D. Acronyms.	1194
	References.	1194

1. Introduction & motivation

Copying existing code fragments and pasting them with or without modifications into other sections of code is a frequent process in software development. The copied code is called a *software clone* and the process is called *software cloning*. A bug detected in one section of code therefore requires correction in all the replicated fragments of code. Thus, it is important to find all related fragments throughout the source code. Considering the high maintenance cost, software clone detection has emerged as an active research area. Different programming paradigms and languages have led to number of clone variants and detection techniques.

There are two landmark literature surveys by Roy and Cordy [187] and Koschke [135] in the field of software clones. However, the volume of research in the field is continually increased. This has led to a need for critical evaluation and integration of the available research in particular the need for a systematic literature review. Kitchenham and Charters [127], Brereton et al. [25] and Budgen and Brereton [30] define a systematic literature review as a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest. This paper reports a systematic literature review to analyze and report the findings in clone-based research.

Systematic literature reviews are time consuming but provide transparent and comprehensive view of ongoing research, and can be used to identify a number of research avenues. This review identifies different key areas of research on software clones, discusses the concepts, research method used and major findings.

1.1. Motivation for work

- Software clones are present in many different software artifacts. Therefore, our study on detection techniques goes beyond source code.
- Upon assessing state of art in software clone research, we realized the lack of systematic literature review. Thus we summarized the existing research based on extensive and systematic database search and report the research gaps for further investigation.

The remainder of this article is structured as follows: Section 2 presents the background, definitions and general theories on software clones. Note, a glossary of acronyms used in this paper can be found in Appendix D. Section 3 describes the research questions, research method that we used to select and review the data material for our research, and presents our chosen framework for anal-

ysis. Section 4 presents the results of the systematic review. In Section 5, we discuss the findings and their implications. For research, we identify what we believe are the most important research gaps. Section 6 concludes and provides recommendations for further research in the area of software clone detection.

2. Background

Firstly, we define the different types of software clones, and the factors leading to software clones. We then summarize the drawbacks of software cloning and state some points as to why software cloning is beneficial sometimes.

2.1. Software clones

The Merriam-Webster dictionary defines a clone as one that appears to be a copy of an original form, thus being synonymous to a duplicate. In source code and other software artifacts, the original (code) fragment is copied and pasted with or without modifications. The pasted (code) fragment is said to be a clone and this activity is known as (code) cloning. However in software engineering field, the term code clones is still searching for a suitable definition. Its vagueness was properly reflected in Ira Baxter's words:

"Clones are segments of code that are similar according to some definition of similarity".

Roy and Cordy [187] mentions code duplication or cloning as a form of software reuse. Baker [7] after experimenting with a sample program concluded that code can shrink by 14% based on exact matches, and 61% based on parameterized matches. The study suggests that as much as 20–30% of large software systems consist of cloned code. Fowler et al. [61] mentions duplication of code as one of the bad practices in software development increasing maintenance cost. The increasing use of open source software and its variants also increased code reuse. Existing code can be modified to cater to new requirements thereby facilitating and advancing open source development. Urgent need to detect software clones has invigorated software clone detection as an active research area.

2.2. Types of clones

It is quite pertinent to mention that standards in case of nomenclature are still missing thereby leading to different taxonomies by different researchers. We list here basic types of clones [20,135,187].

Type 1 (exact clones): Program fragments which are identical except for variations in white space and comments.

Type 2 (renamed/parameterized clones): Program fragments which are structurally/syntactically similar except for changes in identifiers, literals, types, layout and comments.

Type 3 (near miss clones): Program fragments that have been copied with further modifications like statement insertions/deletions in addition to changes in identifiers, literals, types and layouts.

Type 4 (semantic clones): Program fragments which are functionally similar without being textually similar.

Structural clones: These are patterns of interrelated classes emerging from design and analysis space at architecture level. Structural Clones [14] reflect design level similarities which help in maintenance.

Function clones: The clones which are limited to the granularity of a function/method or procedure. Several studies devised the clone detection methods that found the clones at function level which can be extracted in a different procedure.

Model based clones: Nowadays graphical languages are replacing the code as core artifacts for system development. Unexpected

overlaps and duplications in models [47] are termed as model based clones.

2.3. Why clones

Although cut–copy–paste–adapt techniques are considered bad practices from a maintenance point of view, many programmers use them. We list some of the reasons of software cloning.

- *Programmers limitation and time constraints:* The software is seldom written under ideal conditions. Limitations of programmer's skills and hard time constraints inhibit proper software evolution [122]. Only way out is copying/pasting/editing.
- *Complexity of the system:* The difficulty in understanding large systems only promotes copying the existing functionality and logic.
- *Language limitations:* Kim et al. [122] conducted an ethnographic study on why programmers copy and paste code. They concluded that sometimes programmers are forced to copy and paste code due to limitations in programming languages. Many languages lack inherent support for code reuse, leading to duplication.
- *Phobia of fresh code:* Programmers often fear to bring in new ideas in existing software. They fear that introduction of new code may result in a lengthy software development life cycle. Furthermore, it is easier to reuse the existing code than to develop a fresh solution since new code may introduce new errors [99,152].
- *Lack of restructuring:* Programmers delay restructuring (refactoring, abstraction, etc.) of code due to time limits. Often, restructuring gets delayed until after product delivery which increases subsequent maintenance costs. [141].
- *Forking/templating:* Forking is the reuse similar solutions, with the hope that evolution of the code will occur independently at least in short term [120]. Use of structural and functional templates are often mentioned as reuse mechanisms.

2.4. Advantages of clones

Sometimes software developers intentionally introduce code clones into existing software. The study by Kapser and Godfrey [118,120] discusses this issue. Some of the points are mentioned below:

- It is a fast and immediate method of addressing change requirements.
- Some programming paradigms encourage the use of Templates in programming.
- If a programming language lacks reuse and abstraction mechanism, it is the only way left to quickly enhance the existing functionality.
- The overhead of procedure calls sometimes promotes code duplication for efficiency considerations.

2.5. Disadvantages of clones

- *Higher maintenance costs:* Two studies [170,172] confirm that presence of code clones in software greatly increase the post implementation maintenance (preventive and adaptive) effort.
- *Bug propagation:* If a code fragment contains a bug and that fragment is pasted at different places, the same bug will be present in all the code fragments. So code cloning increases the probability of bug propagation [104,158].
- *Bad impact on design:* Code cloning discourages the use of refactoring, inheritance, etc. [61,147]. It leads to bad design practice.

- **Impact on system understanding/improvement/modification:** It is quite common that the person who developed the original system is not the one who is maintaining it. Moreover the presence of duplicated code not only complicates the design but leads to decreased understanding thereby hampering improvements and modifications. In the long run, the software may become so complex that even minor changes are hard to make [136,170].
- **Strain on resources:** Code cloning increases the size of the software system thereby putting a strain on system resources [104,135]. It degrades the overall performance in terms of compilation/execution time and space requirements.

3. Review method

The systematic review reported in this paper was done following the guidelines of Kitchenham et al. [127,128,25]. The steps included in the review include: development of a review protocol, conducting the review, analyzing the results, reporting the results and discussion of findings.

3.1. Planning the review

The review protocol includes the research questions framework, the databases searched, methods used to identify and assess the evidence. Conducting the review comprises identification of primary studies, applying inclusion and exclusion criteria and synthesizing the results. To reduce researcher bias, the protocol, described in the remainder of this section, was developed by one of the authors, reviewed by the other authors and then finalized through discussion, review, and iteration. Electronic databases were extensively searched and its studies are reported. Moreover, some of the leading software engineering journals and conference proceedings which fail to come in electronic search were searched manually. In total 2039 articles appeared in electronic and manual search as shown in Table 2. Study selection procedure is shown in Fig. 1.

3.2. Research questions

The main goal of this systematic review was to identify and classify the existing literature focusing on clone detection, clone management, semantic clone detections and model based clone detection techniques. To plan the review, a set of research questions were needed. Table 1 list the specific research questions and sub questions.

3.3. Sources of information

A broad perspective is necessary for an extensive and broad coverage of the literature. Before starting the search, an appropriate set of databases must be chosen to increase the probability of finding highly relevant articles. [25,127] recommends searching widely in electronic sources and following databases were searched:

- ACM Digital Library (www.acm.org/dl).
- IEEE eXplore (ieeexplore.ieee.org).
- ScienceDirect (www.sciencedirect.com).
- Springer (www.springerlink.com).
- Wiley Interscience (www3.interscience.wiley.com).
- Scientific Literature Digital Library and Search Engine (<http://citeseerx.ist.psu.edu/>).

3.3.1. Additional sources

- Reference lists from primary studies and other review articles.
- Books and Technical Reports.
- Code clone literature website: <http://students.cis.uab.edu/tair-asr/clones/literature/>.

These databases are highly relevant as far as software clone research is concerned. Refs. [127,128] recommend looking for studies in related disciplines for rigorous search. Many developers of tools/algorithms are queried for additional information. The main goal is not to limit the coverage but to make systematic review goal ori-

Table 1
Research question, sub question and motivation.

Research question	Motivation
(1) What is the current status of clone detection?	It helps in understanding the clone detection techniques. Various intermediate representations and match detection techniques used in clone detection technique/tool are reported. Various tools/techniques for clone detection developed till date are mentioned with their share of usage. The research question explores the studies which evaluated/compared different clone detection technique. We mentioned studies which empirically compared different clone detection tools. The number of studies for each type of clone is also reported
(1.1) What methods of clone detection (intermediate representation and match detection technique) are used and what granularity of clone do they use?	
(1.2) What tools are available for software clone detection, what method do they use, what clone type do they address, how frequently are they cited?	
(1.3) Which studies have evaluated methods/tools and with what results?	
(2) Research status in semantic and model based clone detection	
(2.1) What are different studies in semantic clone detection and their comparative analysis?	It is hard to detect semantic clones. Functional equivalence problem is undecidable in general and subgraph isomorphism is NP-complete. Model based clones are quite hard to locate too. So any research indication for the same is immensely helpful. It will help in devising better and highly scalable strategies
(2.2) What are the different studies in model based clone detection and comparative analysis?	
(3) Key sub areas	It helps in knowing the type of study carried out in the article. It is important to know the number of studies for each sub area which helps in identifying key areas for further research. A time based count shows how the key area has evolved over time
(3.1) What are the important areas related to software clones, number of studies in each classified area and their findings?	
(3.2) A time based count to show how the area has evolved over time.	Clone management has turned out to be a cross cutting topic. Recent research is shifting towards efficient clone management techniques. The research questions focuses on understanding the current status of research in clone management and its sub-topics like clone visualization. Different key area identified in research question 3 touch clone management. It is important to know different clone detection methods and clone detection tools overlapping with clone management topics
(4) What is the current status of clone management	
(4.1) What are the studies discussing benefits of clone management?	
(4.2) What is the current status of research in clone management and visualization?	
(4.3) A systematic map showing cross cutting nature of clone management	
(5) What is the subject system used	It will help in building the database on which the clone detection research can be carried out. It is a step towards benchmarking and standardization of comparative analysis studies
(5.1) What is the size of software used in LOC?	
(5.2) What is the programming language of the subject system and whether the system is open source or commercial?	

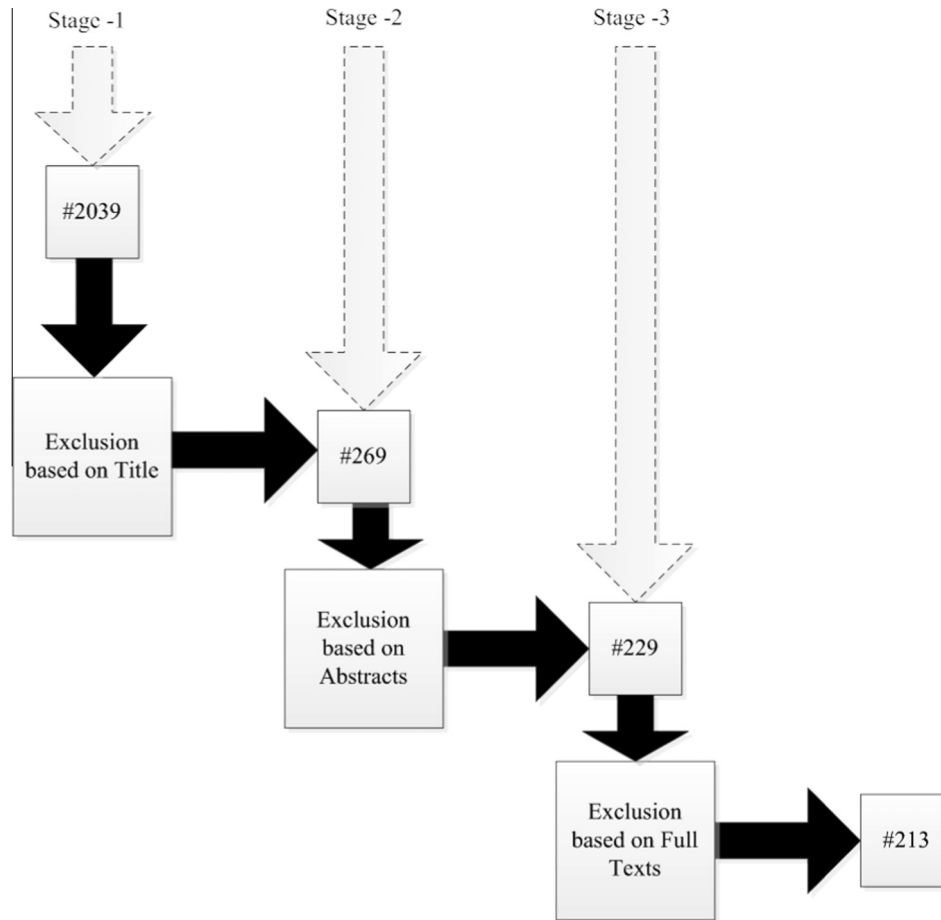


Fig. 1. Study selection procedure.

ented and such that the future directions are clear for further research.

3.4. Search criteria

In almost all the searches, the keyword “clone” is included in abstract. It is an extensive and time consuming process. Table 2 shows the defined search strategy from different e-resources. We tried to extract as much of relevant literature as possible.

We undertook a meticulous database search to ensure the completeness of our study. Even so some of the known research papers were not included in predefined search strategy due to number of reasons. It may be due to different article title, search string not in abstract, etc. It is a hard fact that research community is still striving for standard definition of the term “clone”. Due to this, many traditional research studies refer to clones as duplication of code, etc. These studies are included in the database by keyword search to make review process complete. Redundancy, duplication, copy

Table 2
Search strings.

Sr. no.	E-resource	Search string	Dates	Product/content type	Subjects	#
1	ieeexplore.ieee.org	Abstract: Clone	1988–2011	Conferences, Journals and Standards	Computing and Processing (Hardware and Software), general topics for Engineers (Math, Science and Engineering), Engineering Profession	967
2	www.acm.org	Abstract: Clone	All dates	Journal, Proceedings, Transaction and Magazine	All subjects	485
3	www.sciencedirect.com	Abstract: Clone	All Years	All sources	Computer Science, Decision Sciences and Engineering	134
4	www.springerlink.com	Title: CloneAll Text: (Software/Code)	Entire range of publication dates	All sources	All subjects	245
5	www3.interscience.wiley.com	Article Title: Clone Full Text/Abstract: code or software	All dates	Journals, Reference works, Databases	All subjects	208

and paste are some of the keywords which usually replace clone in article title and abstracts.

3.5. Inclusion and exclusion criteria

In the first stage, irrelevant papers were excluded manually based on titles. In our case, the number of irrelevant papers is high as research articles on clones in biology, mathematics and networks are hard to distinguish from software clones in online databases search. In addition, there are many plagiarism detection tools which overlap with software clone detection tools. Studies were eligible for inclusion in the review if their focus of study was software clone detection or software clones in general. Studies of both students and professional software developers were included. The systematic review included qualitative and quantitative research studies, published up to and including 2011 starting from the initial date of the digital library to make the database search comprehensive. Only studies written in English were included. We included technical reports in our study. Fig. 1 shows the exclusion at different stages. Studies were excluded if their focus, or main focus, was not software clones. Research papers repeating in different e-resources and databases were individually excluded to ensure our research database remained normalized. Position papers were excluded from the literature review but some of the position papers showing future directions are mentioned in the conclusion and future work section. Some of the articles were first published in conferences and then extended versions appear in journals. Such preliminary studies were excluded. All missing relevant papers were found manually using references of identified papers. Some of the journals like Journal of Software Maintenance and Evolution: Research and Practice were individually searched to verify the results from electronic keyword search.

As shown in Fig. 1, Our search returned over 2039 total papers, which were narrowed down to 269 papers based on their titles, and 229 papers based on their abstracts. Then, these 229 papers were read entirely to select a final list of 213 papers based on the inclusion and exclusion criterion.

3.6. Quality assessment

After using the inclusion and exclusion criterion to select relevant papers, a quality assessment was performed on the remaining papers. Since the field is eclectic, a large number of different journals and conferences include research papers of our interest. Quality assessment was done in accordance to CRD guidelines as cited by [127], i.e., each study was assessed for bias, internal validity and external validity of the results.

Using the quality assessment as per Appendix A, all of the included papers contain high-quality software clone research, providing additional confidence in the database of selected papers.

In the quality assessment form (Appendix A), the high level question in Section 1 set the basis for screening the study. After the research paper was included, the paper was studied for classification based on questions in Section 2. Then we proceeded to Sections 3 and 4.

3.7. Data extraction

Appendix B identifies the guidelines for data extraction from all the 213 studies included in the systematic literature review. The data extraction form was designed when we started the information gathering process which is sufficient to address the research questions framed.

Quality assessment form in Appendix A sets the basis for inclusion/exclusion criteria of the study. To some extent, our quality assessment form and data extraction form overlapped. When we

started the systematic literature review, we experienced many problems. It was difficult to extract all the relevant data (Appendix B) from many studies. Due to this, it was necessary to contact many researchers to find the required details which we were not able to infer from the research paper.

The data extraction procedure can be summarized as:

- One of the authors reviewed all of the papers and extracted data from all the 213 agreed primary studies.
- To check the consistency of data extraction, another researcher performed data extraction on a random sample of primary studies and the results were cross checked.
- If there were any disagreement when papers were cross-checked, consensus meetings among the authors were used to resolve them.

4. Results

The goal of this study is to investigate the available literature as per the research questions mentioned in Table 1. Out of 213 papers, seventeen are published in leading journals and the rest is published in premier conferences and workshops on software engineering, programming languages and allied areas. Appendix C lists the journals and conferences publishing most clone related research, including the number of papers which report software clone detection as prime study from each source.

It is worth mentioning regarding the publication for that papers on software clone detection are published in wide variety of journals and conference proceedings. We noticed that conferences like International Conference on Software Engineering, International Conference on Software Maintenance, and Working Conference on Reverse Engineering contribute large share of studies. Premier journals like IEEE Transactions on Software Engineering, Journal of Software Maintenance Evolution: Research and Practice, Journal of Systems and Software contribute greatly to our study domain.

A 83% of the studies were published in conferences and workshops and 17% of the literature appeared in journals.

4.1. Current status of clone detection

There are a large number of clone detection tools/techniques. For any technique/tool, the source representation and the match detection technique are most the important characteristics.

4.1.1. Intermediate source representations and match detection techniques

Initially, source code is pre-processed to remove any uninteresting parts (e.g. comments and blank lines). Then suitable transformation techniques are applied to the pre-processed code to obtain an intermediate representation of the code. Intermediate representation is a way of extracting useful information based upon which comparison is done. Clone granularity defines the boundary of comparison. It can be fixed, e.g. function, class, etc. or free, e.g. number of statements. We listed granularity level for each intermediate representation.

Different clone granularity levels apply to different intermediate source representations. We list all of them in Table 3. Abstract Syntax Trees (ASTs) or Parse trees, Source code or text and Regularized tokens are the most frequently used intermediate transformation.

Match detection algorithms are the prominent issue in clone detection process. After a suitable source code representation and granularity level is decided, an appropriate match detection technique is applied to the units of the source code representation. The output is a list of matches with respect to the transformed code. All the match detection algorithms found in our

primary studies are shown in Table 4. We have listed primary studies that discuss each match detection algorithm. The most frequently occurring match detection techniques are Metric/Feature Vector clustering, Suffix tree based token by token comparison, Substring/Subtree/Model comparison and Dynamic

programming. In post-processing, detected clones are screened for false positives manually. Since many software systems contain large duplication, the outcome of the clone detection results is reported using techniques like scatter plots and other forms of visualization.

Table 3

Intermediate representations, relative count, granularity and citations.

Sr. no.	Intermediate representation/transformation technique	Code	#	Clone granularity level	Citations
1	Regularized tokens	S1	16	Set of statements, set of tokens, fragments of sequence diagrams, files, functions	[7,16,17,27,35,68,89,103,105,106,113,152,153,157,199,230]
2	AST/parse tree	S2	28	Number of tokens, lines of source code, set of instructions, code regions, methods, functions, threshold set by user	[3,4,8,19,22,24,27,31,36,40,58,59,67,95,132–134,147,150,153,170,176,196,202,208,214,226,231]
3	Partite sets and vertices	S3	1	Program	[37]
4	Source code/text	S4	16	Functions, methods, threshold as set by user, number of words/lines	[13,41,42,56,102,104,129,146,164,168,179,180,191,207,209,228]
5	Call graph	S5	1	Functions	[35]
6	Vector space representation	S6	3	Methods, blocks	[74,154,210]
7	One dimensional array	S7	1	Fragments of sequence diagrams	[155]
8	PDG	S8	5	A set of statements that can be extracted into a function, threshold as set by user, non-contiguous clones	[63,84,85,130,138]
9	Index and inverted index	S9	1	Set of statements	[149]
10	Sparse, labeled directed graph	S10	3	Fragments of graph	[47,90,182]
11	Recorded parsing actions and lexical information	S11	1	Set of statements	[166]
12	Abstract memory states	S12	1	Procedures	[125]
13	Description logic	S13	1	Functions, methods, control blocks	[202]

Table 4

Match detection techniques.

Sr. no.	Match detection algorithm	Code	#	Clone granularity level	Citations
1	Suffix Tree based token by token comparison	M1	12	Fragments of sequence diagrams, functions, program fragments, number of tokens as set by user, number of words	[7,59,68,105,106,113,134,153,155,157,166,214]
2	Weighted partite matching	M2	2	Number of blocks, program	[37,47]
3	LSI based clustering algorithms	M3	2	Code segments, functions, files	[164,168]
4	Metrics/feature vectors clustering	M4	17	Set of instructions, code regions, functions and methods, Threshold set by user, blocks	[3,4,8,35,42,95,129,132,133,146,147,154,170,176,179,180,196]
5	Fingerprinting	M5	3	Number of lines, set of statements, blocks	[36,104,180]
6	Anti-unification	M6	3	Threshold of tokens set by user, sub expressions	[27,31,150]
7	Hashing/LSH	M7	6	Set of instructions, code regions, files	[22,63,89,176,196,199]
8	FIM	M8	3	Higher level similarities, i.e. ADT, classes	[17,152,226]
9	Program Slicing	M9	2	Procedure, non-contiguous clones	[84,130]
10	DP	M10	8	Methods, set of statements, programs	[8,22,41,67,106,132,147,231]
11	Suffix arrays	M11	4	Function, sequence of tokens	[16,35,103,230]
12	LCS	M12	6	Program, variable size	[67,89,125,191,210,231]
13	ICA	M13	1	Methods, blocks	[74]
14	Associative array	M14	1	Number of tokens, lines of source code	[58]
15	Nearest neighbor	M15	1	Set of statements	[149]
16	Canonical labeling	M16	2	Fragments of graph	[90,182]
17	Substring/subtree/model comparison	M17	9	Set of tokens, number of lines of code	[13,19,24,56,63,85,207–209]
18	k-Length patch matching	M18	1	Threshold as set by user	[138]
19	Partitioning algorithm	M19	1	Different components of the program	[85]
20	Tree kernel	M20	1	Class, method, set of statements	[40]
21	Levenshtein distance	M21	1	Class, method, set of statements	[95]
22	Semantic web reasoner	M22	1	Functions, methods, control blocks	[202]
23	Random testing	M23	1	Lines of source code	[102]
24	Dot plot/scatter plot	M24	2	Lines of source code	[56,228]

Table 5

Tool type, tool/first author name, source representation/match detection technique, number of studies referring it and citations.

Tool/1st author	Method	#	Citations
<i>Text based</i>			
<i>duploc</i>	Source Code, Substring Comparison/Dot Plot Scatter Plot	5	[20,56,57,195,227]
<i>Simian</i>	Source Code, Substring Comparison	12	[13,21,55,139–143,202–204,207]
<i>DuDe</i>	Source Code, Dot Plot/Scatter Plot	2	[83,228]
<i>SDD</i>	Index and Inverted Index, Nearest Neighbor	1	[149]
<i>CSeR</i>	AST, Metrics/Levenshtein Distance	1	[95]
<i>NICAD</i>	Source Code, LCS	6	[129,169,190,191,193,194]
<i>EqMiner</i>	Source Code, Random Testing	1	[102]
<i>Johnson</i>	Source Code, Fingerprinting	1	[104]
<i>Cordy</i>	Source Code, DP	1	[41]
<i>Marcus</i>	Source Code, LSI	1	[168]
<i>Barbour</i>	Source Code, Substring Comparison	1	[13]
<i>Token based</i>			
<i>dup</i>	Tokens, Suffix Tree	7	[7,20,57,60,68,195,227]
<i>CCFinder (X)</i>	Tokens, Suffix Tree	54	[2,13,15,20,21,24,29,32,38,57,62,66,75–77,82,83,94,95,98,101,112–114,116–120,123–125,137,158,160,165,172–174,186,197,199,201–204,219,223,224,227,230,232,234,235]
<i>D-CCFinder</i>	Tokens, Suffix Tree	2	[156,157]
<i>RTF</i>	Tokens, Suffix Array	1	[16]
<i>clones/cscope</i>	AST, Suffix Tree	1	[134]
<i>iClones</i>	Tokens, Suffix Tree	2	[68,70]
<i>CP-Miner</i>	Tokens, FIM	3	[76,100,152]
<i>SHINOBI</i>	Tokens, Suffix Array	1	[230]
<i>FCFinder</i>	Tokens, LSH	1	[199]
<i>Jian-lin</i>	Tokens, Suffix Array	1	[103]
<i>Chilowicz</i>	Tokens/Call Graph, Metrics/Suffix Array	1	[35]
<i>Tree based</i>			
<i>Deckard</i>	PDG/Parse Tree, LSH/Subtree Comparison	10	[55,63,64,100,109,125,151,184,220,221]
<i>CloneDR</i>	AST, LSH/DP	9	[20,22,32,60,75,206,212,217,227]
<i>SimScan</i>	AST, Subtree Comparison	11	[6,13,21,54,55,60,87,158,174,208,223]
<i>Asta</i>	AST, Associative Array	1	[58]
<i>Clone Digger</i>	AST, Anti-unification	2	[31,40]
<i>Sim</i>	Parse Tree, DP/LCS	1	[67]
<i>ClemanX</i>	AST, Metrics/Feature Vector Clustering/LSH	1	[176]
<i>JCCD API</i>	AST, Subtree Comparison	1	[24]
<i>ccdml</i>	AST, Subtree Comparison	3	[19,29,223]
<i>CloneDetection</i>	AST, FIM	1	[226]
<i>cpdetector</i>	AST, Suffix Tree	1	[134]
<i>clast</i>	Parse Tree, Suffix Tree	1	[59]
<i>Chilowicz</i>	AST, Fingerprinting	1	[36]
<i>Saebjornsen</i>	AST, Metrics/Feature Vector Clustering/LSH	1	[196]
<i>Tairas</i>	AST, Suffix Tree	1	[214]
<i>Lee</i>	AST, Anti-unification	1	[150]
<i>Yang</i>	Parse Tree, DP/LCS	1	[231]
<i>Brown</i>	Tokens/AST, Anti-unification	1	[27]
<i>Graph based</i>			
<i>PDG-DUP</i>	PDG, Program Slicing	2	[29,130]
<i>Scorpio</i>	PDG, Program Slicing	1	[84]
<i>Duplix</i>	PDG, k- length patch matching	3	[20,138,227]
<i>Choi</i>	Partite Sets and Vertices, Weighted Partite Matching	1	[37]
<i>Horwitz</i>	PDG, Substring Comparison/Partitioning Algorithm	1	[85]
<i>Metrics based</i>			
<i>CLAN/Covet</i>	AST, Metrics	10	[4,20,32,34,51,60,144,170,195,227]
<i>Li</i>	Vector Space Representation, Metrics/Feature Vector Clustering	1	[154]
<i>Kontogiannis</i>	AST, Metrics/Feature Vector Clustering/DP	2	[132,133]
<i>Similar Methods Classifier</i>	AST, Metrics/DP	1	[8]
<i>Antoniol</i>	AST, Metrics	1	[3,4]
<i>Dagenais</i>	Source Code, Metrics	1	[42]
<i>Kodhai</i>	Source Code, Metrics	1	[129]
<i>Patenaude</i>	Source Code, Metrics	1	[179]
<i>Perumal</i>	Source Code, Metrics/Fingerprinting	1	[180]
<i>Lavoie</i>	AST, Metrics/DP	1	[147]
<i>Lanubile</i>	Source Code, Metrics/Feature Vector Clustering	1	[146]
<i>Model based</i>			
<i>CloneDetective/ConQAT</i>	Tokens, Suffix Tree/DP	10	[48,49,52,96,105–109]

Table 5 (continued)

Tool/1st author	Method	#	Citations
<i>ModelCD</i>	Sparse labeled direct graph, canonical labeling	2	[49,182]
<i>DuplicationDetector</i>	One Dimensional Array, Suffix Tree	1	[155]
<i>MQ_{clone}</i>	Model/Source Code, Model Comparison	1	[209]
<i>Clone Detective</i>	Sparse labeled direct graph, Weighted Partite Matching	1	[47]
Hummel	Sparse labeled direct graph, canonical labeling	1	[90]
<i>Hybrid Clone Miner</i>	Tokens, FIM	4	[14,17,18,236]
<i>MeCC</i>	Abstract Memory States, LCS	1	[125]
Maeda	Recorded Parsing Actions and Lexical Information, Suffix Tree	1	[166]
Lucia	Source Code, LSI	1	[164]
Li	Tokens/AST, Suffix Tree	1	[153]
Hummel	Tokens, LSH/LCS	1	[89]
Sutton	Vector Space Representation, LCS	1	[210]
Cordy	Vector Space Representation, ICA	1	[74]
<i>DL_Clone</i>	AST/Description Logic, Semantic Web Reasoner	1	[202]
Corazza	AST, Tree Kernel	1	[40]

Table 6

Number of studies referring to different types of clones.

Sr. no.	Type of clone	#	Citations
1	Type 1/exact clones	11	[56,57,59,89,104,129,134,153,157,180,226]
2	Type 2/parameterized clones	23	[7,16,56,57,59,68,83,89,103,113,129,134,148,153,157,166,170,180,214,225,226,230,233]
3	Type 3/near miss clones	27	[22,27,36,40,41,57,59,67,74,83,84,101,106,131,133,134,138,148,150,166,170,179,191,193,203,210,214]
4	Type 4/semantic clones	8	[37,63,102,125,130,138,168,202]
5	Structural clones	3	[14,17,18]
6	Model based clones	7	[47,49,90,105,155,182,209]
7	Function clones	3	[35,58,193]
8	File clones	1	[199]
9	Contextual clones	1	[169]

4.1.2. Clone detection tools

Numerous tools have been developed for clone detection. We have conducted an extensive survey regarding the penetration and usage of clone detection tools for instructors, research or commercial purpose. Table 5 shows tools name/first author name, its source representation and match detection method, which papers have cited it. The table includes clone detection based not only in source code but also on the usage of the tool in other areas like web applications and requirement specifications. As shown in Table 5 we have classified the techniques roughly into seven types: text based, token based, tree based, graph based, metrics based, model and hybrid clone detection techniques. Apart from model-based techniques, these techniques and the tools that use them are discussed in more detail later in this section. Model-based techniques are discussed in more detail in Section 4.2.2.

Different types of clones are detected by different techniques. Many tools/techniques are able to detect only a subset of clone types. In Table 6, we have enumerated the type of clones and identify the studies discussing each type of clone. Table 6 makes it clear that clone research has concentrated primarily on clones of Types 2 and 3, clones of Type 1, Type 4 and Model based clones have been given some attention, but the remaining four clone types have been the subject of very little research.

4.1.2.1. Text based clone detection techniques. In text based clone detection techniques, two code fragments are compared with each other in the form of text/strings/lexemes and similar fragments are reported as code clones. Johnson [104] applied a fingerprinting technique for comparison of source code. Ducasse et al. [56] developed a language independent clone detection tool *duploc* which required no parsing. Line based comparison is done using dynamic pattern matching and results are displayed in the form of dot plot.

DuDe [228] is another line based clone detection tool which is able to detect duplication chains consisting of a number of smaller size exact clones. *Simian* [207] is able to detect clones in different programming languages. If *Simian* does not recognize programming language of the source file, then it treats it as a plain text file to find clones. The *SDD* (Similar Data Detection) [149] tool is helpful in detecting code clones in large size systems. The technique is based on generating index and inverted index for code fragments and their positions. Then an *n*-neighbor distance algorithm is used to find similar fragments.

NICAD [188,191] is a text based hybrid clone detection tool which is able to detect type 3 clones effectively. It is based on a two stage process, viz. identification and normalization of potential clones using pretty printing and code normalization and code comparison using longest common subsequences. Variants of *NICAD* [190] have been used to calculate recall and precision by applying a mutation/injection based framework. *NICAD* has been used to detect function clones in open source systems written in Python to discover any changes in cloning patterns as compared to traditional software systems [194]. Martin and Cordy [169] use *NICAD* to detect and analyze similar web services in the form of contextual clones. Cordy et al. [41] detected near-miss clones in HTML web pages using island grammar to identify and extract all structural fragments and applying UNIX diff as comparator. Barbour et al. [13] used the Knuth–Morris–Pratt algorithm for string comparison to update the clone information from the server incrementally to save time in a client server setup. Later on, only relevant clones are retrieved by individual developers. The technique is found to be faster than string based *Simian* and AST based *SimScan*. However, since it uses string based technique, it fails to detect clones with minor and major changes.

4.1.2.2. Token based clone detection techniques. It is more meaningful in parameterized clone detection as tokens are better than simple keyword matches. In token based clone detection techniques, firstly, tokens are extracted from the source code by lexical analysis. Then some set of tokens (at a specific granularity level) is formed into a sequence. Suffix tree or suffix array based token by token comparison is the heart of token based clone detection algorithms. This match detection technique is the most frequently cited in the literature and was used in one of the first clone detectors *dup* [7]. Token sequences are fed into a suffix tree. The approach used “functor” as an abstraction of concrete values of identifiers and literals that maintains their order. The study reported a greater number of parameterized matches than exact matches in the same subject system. *CCFinder* [113], a token based clone detection tool uses suffix tree matching algorithm to find identical subsequences. It is a popular tool among researchers and has been widely used for code clone analysis, code clone management, etc. Many researchers have worked to enhance the output of *CCFinder* by devising clone visualization tools. For instance, *CCFinder* is used by Basit et al. [15] to study patterns of clones in a standard template library (STL). In their work, they increased the threshold in *CCFinder* to filter small clones. Livieri et al. [157] developed a distributed version of *CCFinder*, viz. *D-CCFinder* for large systems by using 80 workstations in master slave configuration. *CCFinderX* [114] was used to study code clone genealogies at release level. Two studies [197,137] used it for analysis of the relations between open source software quality and code cloning. Monden et al. [173] concentrated on detecting type 2 clones using *CCFinderX*. The number of tokens of largest clone pair and the percentage of duplication within the most suspicious source file pair are important metrics in distinguishing Type 2 clones in their study.

CP-Miner [152] is a token-based tool using frequent itemset mining to detect bugs in the software induced by cloning. *Clone Miner* detects structural clones which are high level abstractions. *RTF* [16] works by applying suffix array on tokens. Suffix array on source code tokens is also used by Jian-lin and Fei-peng [103].

Koschke et al. [134]’s tool *clones* uses a parser and generates abstract syntax tree. Then the AST is serialized and input to suffix tree. The technique is able to detect syntactic units which are not possible by applying suffix tree only. Göde and Koschke [68] developed the incremental tool *iclones* by extending *clones* to detect clones for multiple versions. Li and Thompson [153] used two techniques, viz. combination of token stream and the AST to detect and remove code clones. The two representations are used for their accuracy and speed. Match detection is done with the help of a suffix tree. Another modern multi-input open source clone detector framework *ConQAT* [105,106] detects clones using a suffix tree on tokens. The tool is based on a pipelined approach for extensible token based clone detection. *ConQAT* has been used to detect behaviorally similar code [109]. A number of approaches [47,90] using *ConQAT* have been proposed to detect duplications in Matlab/Simulink models. Acceptance of tools like *CCFinder*, *dup*, *ConQAT* in academia and research showed the usefulness of the fast speed with which suffix trees detect clones.

Yamashina et al. [230] proposed a novel clone detection/modification tool to support the software maintenance process. The study reported a substantial difference between novice and experienced programmers regarding motivation and behavior in handling clones. Using *CCFinderX*’s preprocessor, tokens are gathered from the source code. Then they are input to a suffix array for fast retrieval. Clone retrieval and ranking is performed by the *SHINOBI* server. The evaluation shows *SHINOBI* to be fast and accurate. In the pre-experiment, a large number of programmers were interviewed to analyze their behavior. *SHINOBI* still needs improvement in ranking algorithm. However, it can be extended to support other useful information in addition to code clones.

Sasaki et al. [199] developed a new token based clone detection tool *FCFinder* to detect file clones (files which are copied across projects) using hashing. The study detected 68% of the FreeBSD Ports collection as file clones. However, *FCFinder* took a long time to detect file clones in the 10 GB collection.

4.1.2.3. Tree based clone detection techniques. Abstract syntax trees and parse trees are frequently used representations when source code is to be transformed into tree structures. However, tools based on this approach suffer from large execution times when analysing a large source code base. The output is purely syntactic units of source code which are ready for refactoring. Tree based clone detection is capable of detecting clones in which the code is inserted or deleted (type 3 clones).

Yang [231] proposed one of the first approaches for finding the syntactic differences between two versions of the same program. The technique was based on grammar and builds a variant of a parse tree for both the versions. Detection is applied synchronously to both the trees and is based on the longest common subsequence method of dynamic programming. A limitation of this approach is that his differential comparator can only work for syntactically correct programs conforming to the grammar.

Semantic Designs’ *CloneDR* [22] is another tool which is able to detect exact and near miss clones using hashing and dynamic programming. The tool has different variants for different programming languages. The study reported the use of clone detection in finding commonalities in the form of domain concepts in source code which will help analysts in understanding the design of the system for better maintenance. *SimScan* [208] and *ccdiml* [19] are variations of *CloneDR*. *ccdiml* transforms the source to intermediate representation and *SimScan* applies subtree comparison on the parsed source code. The source code is parsed with the help of ANTLR parser generator. *SimScan* and *ccdiml* have been used to classify the evolution of source code clone fragments in Java and C source code files [223]. Falke et al. [59] and Tairas and Gray [214] used suffix tree to detect clones in code transformed into an AST. The technique has advantage of precision of syntax tree and high speed of suffix tree.

Gitchell and Tran [67] developed *Sim* which converts source programs to parse trees. Viewing parse trees as strings, the tool applied longest common subsequence and dynamic programming to assess similarity. *Deckard* by Jiang et al. [100] is based on computing characteristic vectors from the AST and clustering vectors which are close in Euclidean space by locality sensitive hashing. *Deckard* has been used to identify refactoring on parts of a clone in open source systems [220] and localizing the representation of clone groups [221] and has been used to detect behaviorally similar code [109]. Another application of *Deckard* is to assess the impact of code clone on defects in source code [184]. *Asta* [58] is an AST based tool which works on the phenomenon of structural abstraction of arbitrary sub trees of an AST. *ClemanX* [176,177] is an incremental AST based framework. The tool constructs characteristic vectors from AST subtrees and used locality sensitive hashing. Saebjornsen et al. [196] also used the same set of techniques to detect clones in assembly code.

Anti-unification is used in three studies [150,27,31] to calculate the distance between two AST’s and grouping the similar classes in one cluster. Anti-unification helps to discover common sub-expressions in source code represented as a tree. *CloneDigger* [31] is a language independent tool in which anti-unification is applied to XML representation of source code. *CloneDetection*, a tree based tool by Wahler et al. [226], is based on frequent itemset mining applied on XML representation of source code. Chilowicz et al. [36] developed a new technique to detect exact clones based on syntax tree fingerprinting.

Shifting our focus to code clone management, CSeR (Code Segment Reuse) was developed by Jacob et al. [95] to check copy and paste induced clones in an integrated development environment. The tool was designed to compute clone differences interactively by checking if some piece of code was copy-pasted as the programmer was editing and typing the code. It works on the phenomenon of converting the immediate clone to an AST and computing the difference with the original in a bi-directional manner using metrics like the Levenshtein distance.

Biegel and Diehl [23,24] introduced a novel way for fast and configurable code clone detection using pipelines. They developed JCCD, a flexible and customisable AST based clone detection tool in which several cascaded processors perform various steps of clone detection process. JCCD API parallelizes the detection process using multiple cores.

4.1.2.4. Graph based clone detection techniques. A program dependency graph (PDG) represents control and data flow dependencies of a function of source code. Horwitz [85] used this method to identify syntactic and semantic differences between two versions of a program. Graph representation of source program is partitioned depending upon the behavior of source code fragment. The partitioning algorithm and substring comparison are used to detect similarity. Duplix [138] works on the k-limiting approach of finding maximal similar subgraphs. PDG-DUP [130] is also a PDG-based tool which uses program slicing to find isomorphic subgraphs. It helps in detecting non-contiguous clones. Scorpio [84] by Higo and Kusumoto applied two-way slicing to detect clones. The tool currently works on for Java and is based on number of PDG specializations for Java language and heuristics to speed up the overall detection process. It was developed to address the problem that the existing PDG based clone detection approach is slow in detection of contiguous clones.

4.1.2.5. Metrics based clone detection techniques. In the data extraction form, we extracted the studies using metrics and characteristic features in the same column. Both the match detection techniques are applied after the source code is represented in some suitable form such as an abstract syntax tree. In the systematic literature review, we found a large number of articles using this technique. Characteristic vectors are usually applied for the sub tree matching in an abstract syntax tree or parse tree to detect type 3 clones. The metrics which are extracted from source code are compared to assess similarity. Mayrand et al. [170] developed CLAN, one of the first approaches to compare metrics obtained from an AST of source code. Metrics are calculated from names, layout, expressions and control flow of functions. CLAN has been widely used in the last decade. Patenaude et al. [179] detected metrics from source code organized in five themes, viz. classes, coupling, methods, hierarchical structure and clones. Kontogiannis et al. [132,133] report a technique to detect code clones using metrics extracted from an AST representation of code. Match detection is done by applying dynamic programming on source code lines using minimum edit distance. Balazinska et al. [8] found metrics and applied dynamic matching to an AST representation of source code. Metrics were used by Lanubile and Mallardo [146] to detect function clones in web applications. Perumal et al. [180] used metrics and fingerprinting technique to detect clones in source code. Kodhai et al. [129] and Dagenais et al. [42] applied metrics on textual representations of source code. Li and Sun [154] explored a novel approach by viewing source code clones in metric space with coordinate values. The distance between members across same metric space is measured and the distance reflects similarity between code fragments. This study was accurate and scalable but the technique is yet to be verified for different subject systems.

Metrics have been used successfully in clone analysis [10,82], clone evolution [3,4,51], clone visualization [101], etc.

Lavoie et al. [147] presented a novel technique based on graphics processing unit (GPU) algorithms to compute many instances of the longest common subsequence problem on a generic GPU architecture using classic DP-matching. Because the algorithm is parallelized on a GPU using dynamic pattern matching algorithm, it leads to an opportunity of increased performance. The tool is useful to address the problem of finding more false positives compared to metrics-based clone detection methods. It also compared clone identification problem on CPUs and GPUs hardware architectures. The study recommends clone detection techniques using string-matching with suffix trees could take advantage of the GPU algorithm.

4.1.2.6. Hybrid clone detection techniques. Sutton et al. [210] applied an evolutionary algorithm to search for clones in large code bases. The source code is represented as variable size vector. Clustering of similar code fragments is done with the longest common subsequence. Cordy and Grant [74] introduced a technique using an existing information retrieval method, namely independent component analysis (ICA) to analyze vector representations of software methods. Firstly, singular value decomposition is applied on original method token matrix. Then ICA is applied and points in new vector space that correspond to the input data are recognized. The distance between any two vectors is considered a measure of their similarity.

Maeda [166] introduced a technique based on PALEX source code representation. The PALEX source code includes lexical information and parsing actions recorded from the compiler as it processes the source program. The technique is language independent and uses a suffix tree for comparison. The hybrid approach by Corazza et al. [40] uses a tree kernel which is a class of functions for computing similarities among information arranged in tree structure. It works in a recursive fashion on a tree, starting from similarity measure of the nodes and aggregating the results. The technique is language independent and works at method level for Java programs. The results are compared with AST based tool CloneDigger. A disadvantage of this approach is that it takes a substantial time to execute.

Chilowicz et al. [35] developed a technique to detect function clones in source code represented as call graph using suffix array and metrics. The technique starts with collecting tokens using lexical analysis. Basit and Jarzabek [17] developed Clone Miner using frequent itemset mining which works on the output of token based clone detection tool, RTF [16].

Hummel et al. [89] developed a hybrid incremental index-based clone detector which takes input in the form of token sequences. The index is used to lookup for all clones in a single file and allows updating on addition, deletion or modification. The tool is implemented in ConQAT and runs in a pipeline fashion thereby highly scalable, incremental and provides excellent run time performances. In one case study, 100 machines performed clone detection in 73MLOC in 36 min. The authors proposed the use of locality sensitive hashing in the current implementation to support the detection of type 3 clones. The technique is used in code clone detection as well as model based detection.

Fig. 2 shows different clone detection techniques. The techniques are reported with corresponding source representation and match detection technique. The figure shows the amount of research that has been carried out in different clone detection techniques. Table 3 lists thirteen different intermediate representation or transformation techniques which are referred to in Fig. 2 as SX (where X is a number from 1 to 13). Table 4 lists 24 different techniques of match detection which are referred to in Fig. 2 as MX (where X is a number from 1 to 24).

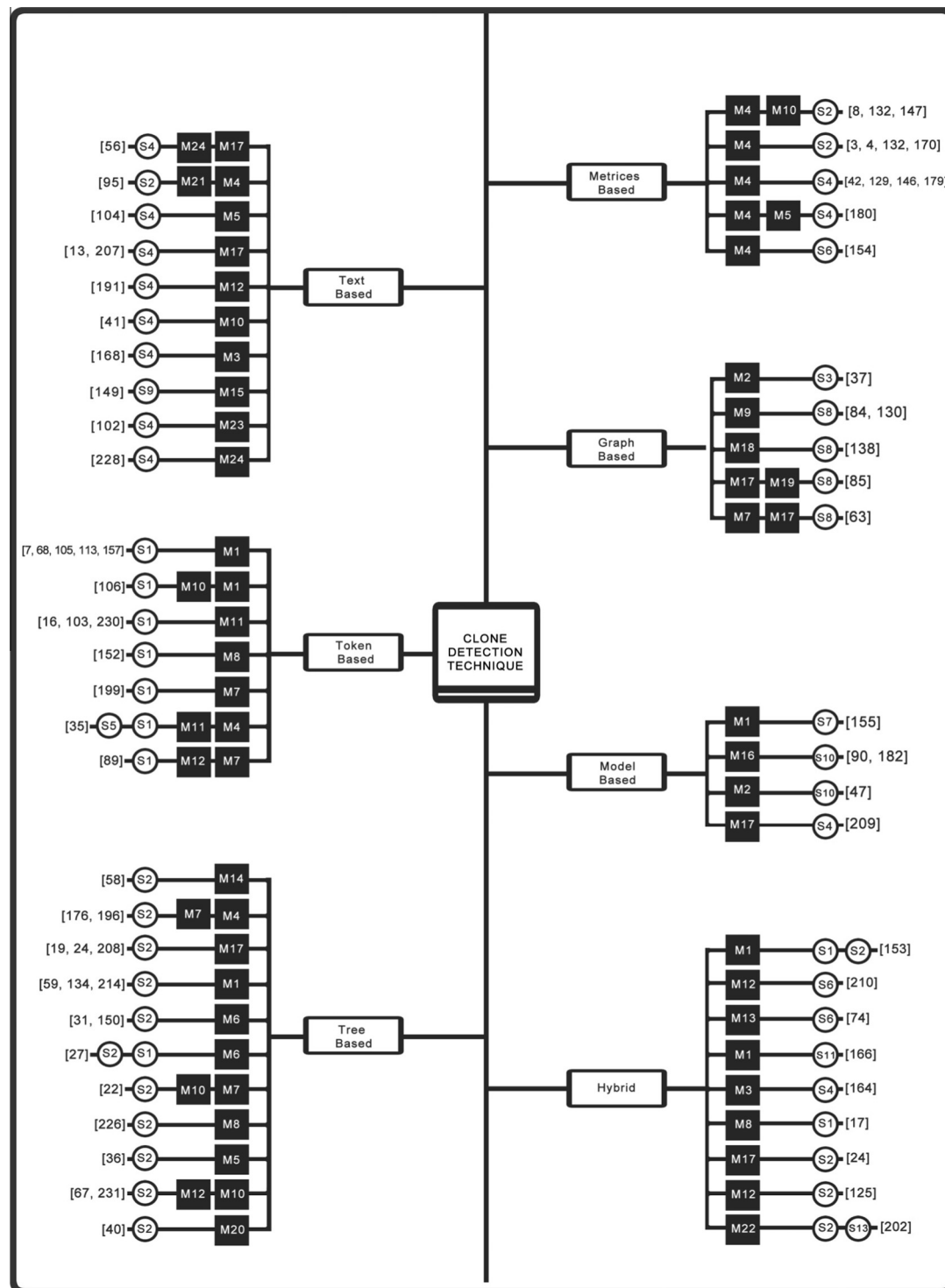


Fig. 2. Different clone detection techniques, match detection algorithms, source representations and citations.

4.1.3. Comparison and evaluation of clone detection tools and techniques

Comparison and evaluation of clone detection techniques is a challenging task. Diverse subject systems and the absence of standard similarity measures complicate the comparison task. So studies covering comparison and evaluation are immensely important to identify an efficient clone detector.

Table 7 shows the empirical studies for comparison and evaluation of clone detection tools. In all the studies barring one by Burd

and Bailey [32], where we mentioned exact values, we wrote ordinal scales (—, +, ++, +++) where — reports lowest and +++ reports highest depending upon the value of parameters in the paper. The reader should refer to the relevant research paper to find the exact values. Different studies evaluated the tools using different parameters. We have left some of the entries blank as the paper does not report the relevant results. These studies are discussed below.

Burd and Bailey [32] conducted the first experiment to compare three clone detection tools, *CCFinder*, *CloneDR* and *Covet* and two

Table 7
Empirical Comparison Studies.

Sr. No.	Reference	Tools Compared	Method	Outcome					
1.	Burd and Bailey [32]			Precision	Recall	No. of Candidates			
		CCFinder	Tokens, Suffix Tree	72	72	77			
		CloneDR	AST, Hashing/ DP	100	9	6			
		Covet	AST, Metrics	63	19	19			
2.	Rysselberghe and Demeyer [195]			Suitability	Relevance	Confidence	Focus		
		duploc	Text, Substring Matching			++			
		dup	Text / Tokens , Suffix Tree			-			
		CLAN	AST, Metrics	++		-			
3.	Bruntink et al. [29]			Error Handling	Tracing	Range Checking	Null Value Checking	Memory Error Handling	
		CCFinder	Token, Suffix Tree	++			++		
		ccdimpl	AST, Tree Matching	++		++	++		
		PDG-DUP	PDG, Program Slicing		++			++	
4.	Koschke et al. [134]			No. of Candidates	Rejected Candidates		True Negatives		Recall
		CCFinder	Token, Suffix Tree	++	++		+		++
		CloneDR	AST, Hashing/ DP	-	+		+++		-
		CLAN	AST, Metrics	-	-		+++		-
		dup	Text / Tokens , Suffix Tree	+	++		++		+
		duplix	PDG, k length patch matching	++	+++		+++		-
		duploc	Text, Substring Matching	+	++		+++		+
		cpdetector	AST, Suffix Tree	+	++		++		+
		ccdimpl	AST, Tree Matching	++	++		++		++
		clones	Tokens, Suffix Tree	+++	+++		++		+
		cscope	Tokens, Suffix Tree	++	+++		++		+
5.	Bellon et al. [20]			Recall	Precision		Time Requirements		Space Requirements
		CCFinder	Tokens, Suffix Tree	++			+		+
		CloneDR	AST, Hashing/ DP		++		-		-
		CLAN	AST, Metrics		++		++		++
		dup	Text / Tokens , Suffix Tree	++			++		+
		duplix	PDG, k length patch matching				--		+
		duploc	Text, Substring Matching	++					
6.	Falke et al. [59]			No. of Candidates		Rejected Candidates		Recall	
		ccdimpl	AST, Tree Matching	+		++		++	
		cpdetector	AST, Suffix Tree	+		++		+++	
		clast	Parse Tree, Suffix Tree	+		++		+++	
		clast-ba	Parse Tree, Suffix Tree	+		++		+++	
		clones-uk	Tokens, Suffix Tree						
		clones-ba	Tokens, Suffix Tree						
		cscope	Tokens, Suffix Tree	+++		+++		+++	
7.	Roy and Cordy [190]			Recall	Precision				
		Basic NICAD	Text, LCS	+	+				
		FlexP NICAD	Text, LCS	+	+				
		Full NICAD	Text, LCS	++	++				
8.	Pham et al. [182]			Precision	Completeness				
		CloneDetective	Labeled multigraph, maximum weighted bipartite matching	+	+				
		ModelCD	Sparse labeled directed graph, canonical labeling	++	++				
9.	Deissen-boeck et al. [49]			Time					
		CloneDetective	Labeled multigraph, maximum weighted bipartite matching	++					
		ModelCD	Sparse labeled directed graph, canonical labeling	+					
10.	Juergens et al. [109]			Partial Clone Recall		Full Clone Recall			
		Deckard	AST, Characteristic Vector/ Hashing	++		++			
		ConQAT	Tokens, Suffix Tree	+		+			
11.	Selim et al. [203]			Recall	Precision	Rejected Candidates			
		Selim et al.	Jimple, Suffix Tree/Subtree Comparison	++	++	++			
		CloneDR	AST, Hashing/ DP	+	+	+			
		CLAN	AST, Metrics	+	+	+			

plagiarism detectors *JPlag* and *Moss*. They validated all the clone candidates of the subject system obtained by all the tools and made a human oracle. Then they used the human oracle for comparing the different techniques in terms of precision and recall. The result of their experiment shows 100% precision for syntax-based technique *CloneDR* indicating that no false positives in its detected clones are reported by this tool, however, recall was very low. The token-based tool *CCFinder* shows the highest recall (72%) and reasonable precision (72%). The metric based tool *Covet* showed the minimum precision (63%) compared to the other tools. The limitation of the case study was in terms of system size. The case study used source code of *GraphTool* written in Java which was only 16 KLOC.

Rysseberghe and Demeyer [195] compared text-based *duploc*, token-based *dup* and metric based clone detector *CLAN*. The goal of this comparative study was to identify the most appropriate clone detection tool for refactoring. They use five small to medium (under 10KLOC) sized cases for evaluating the techniques. These techniques were evaluated qualitatively rather than quantitatively in terms of suitability, relevance, confidence and focus. The results show that metric fingerprint techniques were best suited to work with a refactoring tool. No significant difference was found between the approaches with respect to relevance and focus. Simple line matching gives the highest confidence as it finds exact matches whereas all the other techniques requires a manual inspection to reject false positives. Their study also found that parameterised matching techniques returns more clones.

Koschke et al. [134] evaluated their AST-suffix tree based tool *cpdetector* in terms of precision, recall and runtime against existing tools using Bellon's experimental procedure and additional tools are *cpdetector*, *ccdiml*, *clones* and *cscope*. Since their developed tool is suitable for C systems only, therefore they worked only with 4 C systems of Bellon's experiment and provided detail results for only one system. The results show that the AST-based tool *ccdiml* shows a good recall (53%). However, the average recall for the token-based tools is almost double (54%) than the AST-based tools (30%). Their experiment result shows that clones found 71% more clones than *CCFinder*. Metrics based tool *CLAN* is good and *duplix* is worst in terms of runtime.

Bellon et al. [20] conducted a tool comparison experiment with the same three clone detection tools that were used in Burd and Bailey's study and with three additional tools, viz. *dup*, *duplix* and *duploc*. They also used software systems in Java and C (four Java and four C systems) totalling almost 850KLOC. Their experiment showed that precision and recall was complementary for each of the tool except the PDG-based *duplix* where both as attributes exhibited the lowest values. The AST based *CloneDR* and the metrics based *CLAN* had high precision but their recall was very low. The token based tools (*dup* and *CCFinder*) and the text based tool *duploc* had the highest recall but low precision. The experiment shows that the PDG-based tool *duplix* performed very badly in terms of execution time compared to the other clone detection tools.

Falke et al. [59] empirically compared *ccdiml*, *cpdetector*, *clast*, *clast-ba*, *clones-uk*, *clones-ba* and *cscope* using subject systems in C and Java. The results show that token based tools yield large number of clones. AST matching shows lower rejection rate, but also has a lower recall. The result indicates that detection based on suffix trees is faster than detection based on tree matching. Parse tree based tools show lower rejection rate than AST-based tools for C systems but a higher rate for Java systems. Also parse tree based tools are faster than AST based tools for java systems. In contrary to previous experiments, this study does not show high recall for token based tools. According to this study the advantage of token based techniques is that it is easier to implement lexer than parser and requires less space than AST. On the other hand

AST based techniques are good to find syntactic clones and help to filter those syntactic structures which are of little interest.

Roy and Cordy [190] propose a mutation/injection based framework for evaluating clone detection techniques. In this method mutant versions of code fragments are created. Then these mutant versions are injected into original source code. Different variants of tool *NICAD* (*Basic NICAD*, *FlexP NICAD* and *Full NICAD*) are run on these mutant versions to compare tool on basis of precision and recall. The result of this study shows that *Basic NICAD* has poor recall for clones generated by mutant operators of type-2 clones. *FlexP NICAD* is also not good to find clones generated by mutant operators of type-2 clones. *Full NICAD* is best to find clones generated by mutant operators of all type of clones. The advantage of this framework is to evaluate and compare recall of different clone detection tools without manual intervention and similarly precision can be evaluated either automatically or by using an interface with minimal manual intervention.

The experiment [49] compared the techniques of Pham et al. [182] and Deissenboeck et al. [47]. The study proposes reduction in branching to avoid multiple occurrence of sub graph in the search tree. This is done to prune the search space to make it relevant and fast. The techniques to remove obvious clones and branch reduction lead to lower recall.

Juergens et al. [109] demonstrated the applicability of state of the art tools in detecting behaviorally similar code. The authors stated that behaviorally similar code is highly unlikely to be syntactically similar and such code results due to independent development. *Deckard* and *ConQAT* cannot detect more than 1% of such code. Selim et al. [203] proposed a hybrid technique in which clone detection is performed simultaneously for source code and intermediate code which are merged to detect near miss clones in Java. Using Bellon benchmark [20], comparison with other state of the art tools is done. The technique gives lower precision and higher recall than *CCFinder* and *Simian*, when used standalone. The technique detected some clones which are not useful. Use of the technique on large code bases is still to be done.

An assessment of Type 3 clones as detected by the clone detection tools namely *ccdiml*, *clones*, *clast*, *duplix* and *CLAN* is performed by Tiarks et al. [222]. The study apparently points out that existing type-3 clone detectors need to be improved as only 25% of detected clones are accepted by human oracle. An empirical study [193] is carried out regarding function clones in open source software.

Upon assessing each of studies, it is difficult to find out the best tool for each study. After a complete review of all studies, we are able to write following general remarks:

Token based clone detection tools like *CCFinder* detected large number of clones. They have high recall and reasonable precision. These tools do not help the developer in refactoring in a straightforward manner.

Tree based tools like *CloneDR* have high precision. Though these tools detect very less number of clones with low recall, but the detected candidates are ready for refactoring thus helping the developer in clone management.

Metrics based tools like *CLAN* has good precision but suffer from low recall and less number of candidates detected. These tools run fast and detect function clones ready for refactoring.

Tools which apply suffix tree in AST representation of code like *cpdetector* works faster than other AST based tools.

One PDG based tool named *duplix* is able to recognize type 3 clones only but suffer from high time complexity.

We have not found any empirical study comparing various semantic clone detection tools/techniques.

There are two papers comparing model based clone detection approaches, i.e. *ModelCD* and *CloneDetective* and *ConQAT*.

Benchmark [20] has been used by [134,20,59,203] in their empirical studies. It consists of clone pairs validated by humans for eight software systems written in C and Java from different application domains.

We have mentioned the empirical studies comparing different clone detection tools in the above paragraphs. These studies are undertaken by taking one or more subject systems to evaluate the tools. Evaluation is done with parameters like precision, recall, etc. as shown in Table 7. Other studies comparing clone detection tools/techniques qualitatively are explored in the following paragraphs. Studies devising clone oracle are also discussed in following paragraphs.

Ducasse et al. [57] compared their string based clone detection technique *duploc* with Baker's token based tool, *dup* and Kamiya's token based tool, *CCFinder* and confirms that this inexpensive clone detection technique can also yield high recall and acceptable precision. This study uses WelTab (9KLOC) and Cook (46 KLOC) as subject systems. The detailed results of the study are shown in Table 8. For *duploc* – means no normalization; C means constants normalized; I means identifiers normalized; F means functions normalized; CI means constants and identifiers normalized; IF mean identifiers and functions normalized; CIF means full normalization; and 0, 1, 2 specifies the maximum gap size in the comparison sequence. The results of the study show that for WelTab, normalizing identifiers and function names is important to achieve similar results as that of *dup* and *CCFinder*. For Cook, normalizing identifiers can lead to too many clones but normalizing constants only with maximum gap size zero gives good precision. The study confirms that this inexpensive clone detection technique can also yield high recall and acceptable precision, although the tool needed to be extensively calibrated to each system. However, the results also show clearly that the effectiveness of clone detection tools is strongly influenced by the specific system to which the algorithms are applied. This confirms that studies attempting to compare different types of clone detection tools must evaluate them on a variety of different systems.

Roy and Cordy [189] compared clone detection techniques on the basis of several criteria like language support, comparison granularity, clone similarity, and code representation. They also propose a set of hypothetical editing scenarios for different clone types and evaluate the clone detection techniques based on their estimated potential to accurately detect clones that may be created by those scenarios. Their studies results shows that hybrid technique based on tree-based techniques (e.g., *cpdetector*) and text based techniques (e.g. *duploc*) can detect type-1 clones efficiently. Hybrid approach of token based (*dup* and *RTF*) and AST based (*cpdetector* and *Asta*) best suits to find type-2 clones. Type-3 clones are best found by hybrid approach based on text based (*DuDe* and *SDD*) and AST based (*Deckard*) techniques and graph based techniques (*duplix* and *GPLAG*) are best suitable to find semantic clones. The results of this study are predictive rather than empirical but assist to understand and find interesting combinations of techniques.

In another study, Roy et al. [192] conducted a large case study to classify and compare clone detection approaches based on a number of facets, each of which has a set of attributes. They qualita-

tively evaluated the classified techniques and tools with respect to taxonomy of editing scenarios designed to model the creation of Type-1, Type-2, Type-3 and Type-4 clones. So this case study compares the clone detection techniques and tools qualitatively and helps to understand the potential of each technique and tool to find clones generated by different scenarios.

Walenstein et al. [227] carried out a study to highlight the problems in devising universal oracle. The study involves researchers' view in classifying the candidate clones into clones and non-clones categories as detected by tools, viz. *dup*, *CloneDR*, *CCFinder*, *duplix*, *CLAN* and *duploc*. Authors observed high level of disagreement. This exploratory study is a step towards clone detector benchmarks. The study pointed out that inter-rater reliability measures should be calculated for human generated reference data. Lakhota et al. [145] stressed on the need to create a community supported open benchmark suite to help researchers in evaluating and comparing clone detectors and choosing standard subject systems for the study. Lavoie et al. [148] presented a novel technique to construct clone oracles automatically in large systems based on the Levenshtein metric. Compared to manual oracles, this oracle is of good quality for type-3 clone detection assessment and is able to deal with large code base in reasonable time. The oracle generates good quality clones and aims for comparison and evaluation of clone detection techniques in a reasonable time.

Barring one comprehensive study by Roy et al. [192], comparative studies of clone detection techniques pertain to some subset of tools. We found only eleven empirical studies as shown in Table 7 which shows apparent lack of work in comparison and evaluation of clone detection techniques. We found two studies by Deisenboeck et al. [49] and Pham et al. [182] on comparison of existing model based clone detection techniques.

4.2. Status of research in semantic and model clone detection techniques

Semantic clone detection and model based clone detection techniques are challenging upcoming areas. We focused our review to analyze the available techniques in detail.

4.2.1. Semantic clone detection

Two program fragments differing in their concrete syntax may be semantically very close. Detecting semantic equivalence is very difficult. It needs deep semantic analysis. There are some studies which tried to detect semantic clones. They are mostly approximations to type 4 clones. In 1990, a key article [85] by Horwitz was published to detect textual and semantic similarities. Table 9 compares and details different semantic clone detection techniques.

PDG is a directed attributed graph representing the statement and control flow and captures semantic information from source code. It works as an abstraction of source program. The application of a subgraph isomorphism technique to detect clones in the form of isomorphic subgraphs is NP-complete. Krinke [138], Komondoor and Horwitz [130], Gabel et al. [63] presented techniques to detect semantic clones using PDG as source representation. Krinke [138] used k-length patch matching to detect maximal induced subgraphs. The technique worked well with reasonable precision and recall on

Table 8
Comparison of three clone detection systems on two software products.

Study/tool	WELTAB			COOK		
	Candidates	Precision (%)	Recall (%)	Candidates	Precision (%)	Recall (%)
Dup	2742	80	80	8593	29	70
CCFinder	3888	99	93	2388	42	43
Duploc	2378 (IF, 1)	90	86	9043 (CIF, 0)	26	71
	2609 (IF, 1)	90	88	7661 (–, 2)	42	64
	3761 (CIF, 0)	91	92	276 (C, 0)	49	26

Table 9
Semantic clone detection and comparative analysis.

	Normalizations/ transformations	Source code representation	Clone matching technique	Advantages	Disadvantages	Tools
Jens Krinke [138]	PDG	Fine grained PDG	n -Length patch matching (maximal similar subgraphs)	High precision and recall	Needs a PDG generator for different language, works for C language	<i>DupliX</i>
Komondoor & Horwitz [130]	CodeSurfer to PDG	PDG	PDG Subgraph matching using program slicing	Mechanical refactoring can lead to procedure extraction	Needs a PDG generator, very slow for large code bases	<i>PDG-DUP</i>
Choi et al. [37]	Programs to partite sets and functions to vertices	Birthmarks	Maximum weighted bipartite matching	Efficient, highly resilient	Needs deobfuscation methods against attacks	
Marcus and Maletic [168]	Comment removal and token regularization	Text	Vector representation using LSI	Finds high level structural clones	Highly dependent on comments, low precision	
Gabel et al. [63]	CodeSurfer to PDG to AST	PDG	Characteristic vectors in Euclidean Space	Highly scalable	Needs a PDG generator, slow for large code bases	<i>Enhanced Deckard</i>
Jiang and Su [102]	Program text to intermediate language	C intermediate language	Automated random testing	Scalable	Works for C programs only	<i>EqMiner</i>
Kim et al. [125]	Semantic based static program analysis tool	Abstract memory states	Abstract memory state comparison	Precise, can be used to identify bugs, inconsistencies, plagiarism	More false positives, semantic based static analyzer takes lot of time	<i>MeCC</i>
Philipp Schugerl [202]	AST to description logic	Description logic	Semantic web reasoner	Scalable, can be parallelized	Fails to detect smaller clones and clones across methods	<i>DL_Clone</i>

sample software systems. Komondoor and Horwitz [130] used program slicing to detect isomorphic subgraphs in the PDG. The technique uses backward and forward slicing and able to detect good clone candidates for procedure extraction. It can detect non-contiguous clones. Gabel et al. [63] presented a scalable technique to detect semantic clones from the PDG representation of the source code. The key element of the algorithm is to map the NP-complete graph isomorphism problem to tree similarity. The tree similarity is based upon comparing characteristic vectors. Upon empirical evaluation, the tool has good execution times on large code bases.

Marcus and Maletic [168] applied latent semantic indexing (LSI) on the textual representation of source code to identify semantic similarities across functions/files/programs. LSI is vector based statistical method to represent meanings of comments and identifiers of source code. They tried to detect similar high level concept clones, e.g. abstract data types.

Software birthmarks have been used successfully to detect copied programs and software theft. Choi et al. [37] used a set of API calls to detect similar programs. The similarity technique depended upon maximum weighted bipartite matching. In this way, the method is useful in detecting semantic equivalences duplications in case of software thefts. However, the technique is vulnerable to deobfuscation attacks.

There is only one study by Jiang and Su [102] which identified functionally equivalent code fragments of arbitrary size depending on the input–output behavior of a piece of code. They detected two pieces of code that always produce same output on random inputs although they are syntactically different. They defined functional equivalence as a special case of semantic equivalence. The results were validated by applying random tests. The tool was scalable and able to work on million lines of code finding that 58% of functionally equivalent code was syntactically different. The technique worked for C programs only.

Kim et al. [125] proposed *MeCC*, a semantic clone detector based on a path-sensitive semantic-based static analyzer. The analyzer was used to estimate the memory states at each procedure's exit point; then memory states were compared to determine clones. The authors compared their findings with *CCFinder* and *Deckard*. *MeCC* is able to detect larger number of procedural clones, to trace inconsistencies, identify refactoring candidates and understand software evolution related to semantic clones.

Schugerl [202] presented a novel technique to detect global clones. An abstract syntax tree representation of the source code is normalized in the form of description logic. Then a semantic web reasoner is applied to trace similar source code based on control-blocks and used data types. The author compared the technique with state of the art clone detection tools but only on Java source code. The technique is highly scalable with the use of semantic web reasoner for match detection.

Several authors suggested areas for further research. Marcus and Maletic [168] proposed the combination of multiple detection algorithms in future. A number of hybrid clone detection algorithms were developed as a result. PDG based approaches of detecting semantic clones suffer from slow generation of clone pairs and imprecise definition of semantic clones. Future work [63] lies in developing the framework to help in fast generation of PDG from source code. However, PDG does not consider statement ordering, thus detecting non-contiguous clones which may turn out to be false positives during manual verification. So, approximate solutions of mapping PDG to trees as by Gabel et al. [63] by applying tree similarity technique are more scalable. Choi et al. [37] proposed both extending birthmarks with more information and making technique robust against attacks. Schugerl [202]'s technique can be parallelized using a cluster of computers. Empirical evaluation across more subject systems would help identify future extensions of the tool which currently does not detect small clones and clones across method. Jiang and Su [102] proposed exploring future research on functionality-equivalent code refactoring and reuse. A general method for different programming languages should be developed to detect functionally equivalent but syntactically different code fragments. *MeCC* [125] can be extended by adapting a static analyzer to collect memory states for any arbitrary code blocks to make clone detection possible at finer granularity.

4.2.2. Model based clone detection

With the rise in abstraction, model driven development has turned to be an emerging area. Large models are developed using UML, Matlab/Simulink, domain specific modeling languages, etc. The presence of duplicated sub structures in different types of models cannot be ruled out. Model based clone detection techniques are still in their infancy. Detecting clones in models is an

Table 10

Model based clone detection and findings.

	Liu et al. [155]	Pham et al. [182]	Deissenboeck et al. [47]	Herald Storrle [209]	Hummel et al. [90]
Preprocessing/normalizations	Two dimensional sequence diagrams into one dimensional array	Transformation of models to graphs, assigning labels to relevant blocks	Transformation of models to graphs, assigning labels to relevant blocks	XMI files from UML domain models	Transformation of models to graphs, assigning labels to relevant blocks
Source representation	One dimensional array	Sparse, labeled directed graph	Labeled multigraph	Prolog code	Directed, labeled multigraph
Clone matching technique	Suffix tree	Canonical matching, vector based approach	Maximum weighted bipartite matching	Model matching	Canonical matching, clone index based hashing
Advantages	High precision and recall	Algorithm is able to detect model fragments with modifications, incremental	Scalable	Supports refactoring	Incremental, distributed, fast detection time
Disadvantages	Works only for sequence diagrams	Lower precision	Large number of false positives	Complex java implementation	Infeasible for large subgraphs
Application area	Sequence diagrams	Matlab/Simulink models	Matlab/Simulink/Targetlink models	UML domain models	Matlab/Simulink models
Model clone granularity	Extractable fragment of a sequence diagram	Number of blocks	Number of blocks	Sub models	Sub models
Tools	<i>DuplicationDetector</i>	<i>ModelCD</i>	<i>CloneDetective</i>	<i>MQ_{clone}</i>	Integrated in <i>ConQAT</i>

emerging area. Many researchers tried to find the clones in models. So this classification includes studies and tools related to detection of duplication in different diagrams and models as summarized in Table 10.

Liu et al. [155] detected duplications in sequence diagrams by converting the 2-dimensional sequence diagram to a 1-dimensional array. Then, the 1-dimensional array is used to build suffix tree. Common prefixes are identified from the suffix tree in the form of reusable sequence diagram as refactoring candidates. The study confirmed the presence of 14% duplication in sequence diagrams of sample industrial projects.

The automatic detection of clones in models leads to identification of potential domain specific library elements [47]. Deissenboeck et al. [47] used *ConQAT* as an integrated framework to detect clones in Simulink/Matlab models especially in automotive domain. The tool *CloneDetective*, which is part of the *ConQAT* framework, works by representing the model as a normalized multigraph where labels are assigned to relevant blocks. Similarity between blocks is checked by a heuristic which performs a depth first search based looking for matched pairs. After detection, clones are clustered based on set of nodes using union function.

Pham et al. [182] presented two algorithms namely *escan* and *ascan* to detect clones in models. These were incorporated in the tool *ModelCD*. Firstly, the model was pre-processed to be represented as a parsed, labeled directed graph. *escan* was used to detect exact matching using an advanced graph matching technique called canonical labeling and *ascan* was used to detect approximate matching by counting vector of sequence of nodes and edges' labels. The technique is incremental in the way it generates candidate cloned sub graphs. Their tool *ModelCD* was compared with *CloneDetective* (which is included in the *ConQAT* framework) [47]. For the same clone granularity and subject systems, both the tools were compared based on four parameters: Precision, completeness, scalability, incrementality. *ModelCD* performed better. In a subsequent paper, Deissenboeck et al. [49] discussed the presence of a large number of false positives in Simulink/Matlab models, pointing out that it is important to identify relevant clones. Model clones suffer from the problem of scalability, clone inspection and relevance. Their study provided useful insights in addressing these problems in real time industrial context. The authors proposed reducing the size of models to make clone detection speedier. Firstly, removal of obvious cloned sub-systems is carried out. After which, as proposed by Pham et al. [182], all nodes with high degree are removed. After the detection process is complete, the algorithm

tries to connect smaller nodes that are connected to each other over high degree nodes. Deissenboeck et al. [49] then compared their enhanced version of *ConQAT* and *escan* showing *ConQAT* has a faster execution times than *escan*.

Storrle [209] pioneered the detection of clones in all types of UML domain models. The technique was based on model querying. Using any of the UML case tools, XMI files are generated from UML domain models. These files are transformed into Prolog files. A small model is input in the query and using model matching, the output is generated. The tool is capable of comparing models originating from diverse sources including a variety of UML versions. We observed that the tool is naïve and still to be verified for different subject systems. Hummel et al. [90] introduced an incremental algorithm for model clone detection. A Simulink/Matlab model is pre-processed by flattening the model into a directed multigraph. Then, relevant edges and blocks are labeled. A clone index is created for all subgraphs of same size. Canonical label of all subgraphs in the clone index is calculated and similar labels are hashed. Clone retrieval and index update are integrated for fast retrieval. It has not been verified on large models.

It is still to be verified whether canonical matching and vector based approach can be applied on other graph based models like UML models. We observed one study by Deissenboeck et al. [49] highlighting the practical issues to be resolved in model clone detection. The study pointed out that ranking of clones may help in improving scalability and relevance of model clones. A comprehensive model clone detection tool for UML models and other data flow languages is missing. Different forms of models have individual features which need to be exploited for clone detection. A mapping of UML constructs from syntactical to semantic domain may help in detecting clones having same behavior but different syntactical structure. We noticed a vagueness in the definition of model clones which hinders understanding of the topic area.

4.3. Key sub areas

We categorized the literature related to software clones in six different yet allied areas. We realized that different identified categories are very important from the research perspective in the field of software clones. Although many sub areas overlap each other, we have tried to separate them into self contained topics. Cross cutting studies in some of these key areas are included in a clone management systematic map constructed to answer research question 4 in Section 4.4.3 (see Fig. 4). Details of these

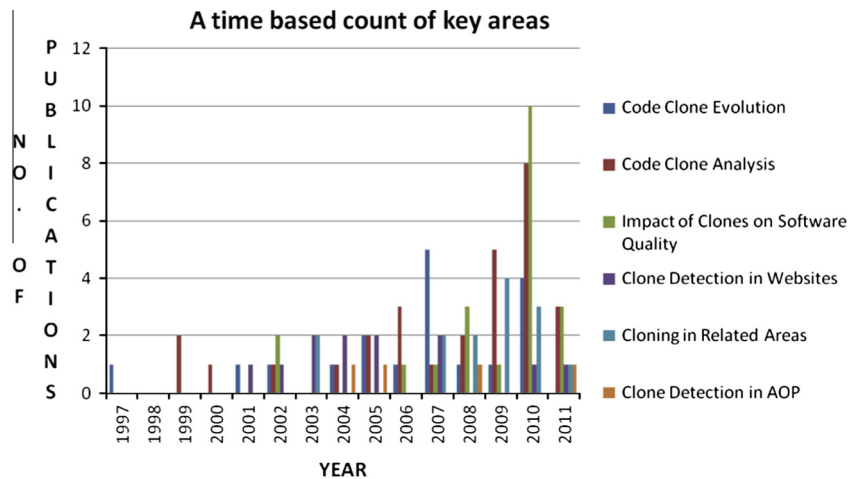


Fig. 3. A time based count for key areas.

studies have been discussed in their respective key areas to make the corresponding key area complete in itself. This is done to help researchers to pursue further research in these sub areas.

4.3.1. Code clone evolution

Code clone evolution depicts the patterns in which the code is developed throughout the history of the software. Other interesting patterns of clones by evolution of software through different versions are also covered in clone evolution.

As software evolves with time and introduction of newer versions, so clones present in the software evolve too. Laguë et al. [144] pioneered clone evolution analysis using metrics based function clone detection technique for large telecommunication monitoring system of 1 million LOC. Two changes, viz. preventive control (to keep the introduction of newer clones in system under control) and problem mining (to cope with the existing code clones in a system under continuous development and maintenance) are studied to assess their impact on clone detection design. Both the changes can effectively improve clone management. Antoniol et al. [3,4] analyzed different versions of the Linux kernel to check the changes in cloning patterns. In an earlier study [3], the author modeled the time series by analyzing clones over several versions of software. Shawky and Ali [206] modeled clone evolution using chaos theory. The study predicted clones in new versions of open source systems with high prediction accuracy. Di Penta et al. devised a framework, i.e. *Evolution Doctor* [50,51] to control software system evolution. It defines a set of methods and tools to deal with removal of clones, restructuring and reorganizing the source code.

Code clone genealogies approximate how programmers create, propagate and evolve code clones [124,126]. Saha et al. [197] carried out an empirical study to evaluate and understand clone genealogies in 17 open source software systems in four different languages at release level. Their study is an extension to a study by Kim et al. [124]. Unlike [124], this study analyzed the evolution of clones at release level. It used *CCFinderX*, a token based clone detection tool. A location independent approach was used to match identifier names across releases. The clone genealogies were classified as alive genealogy, dead genealogy, syntactically similar genealogy and consistently changed genealogy.

Clone Evolution View [11] was developed to work with a metrics based clone detection tool to study how developers copy across different versions of a program. Livieri et al. [156] carried out evolution analysis of 136 versions of Linux kernel using code clone coverage metrics. They used *D-CCFinder*, a distributed extension of code clone detection tool *CCFinder*. Aversano et al. [6] performed an empirical study using *SimScan*, a syntax based clone detector, on

two open source Java systems. They defined three clone evolution patterns to study the effect of maintenance activities on clones. Bakota et al. [12] attempted to connect separate clone instances across different versions of the software based on similarity measure. The clone detection is carried out by the clone detection tool, *clones* [134]. The concept of dynamic code smells is introduced which define the conditions under which a clone becomes suspicious compared to its other occurrences.

Krinke [139] pioneered the argument that clones have a bad impact on software maintenance. In five large open source systems, the author traced the changes across 10 weeks. The study showed that clone groups are continually changed in 50% cases. Lozano et al. [158] analyzed clones at method level. *CloneTracker* was applied to measure the number and density of changes in two cases, i.e. during the presence of cloned code in methods and not. The study observed that cloned code methods need to be changed more frequently than non-cloned code. Lozano et al. [159] presented a way to perform origin analysis and assess the effect of cloning on methods' maintenance effort. The study computed measures of likelihood and the impact of change as they represent the work required for maintaining a method. The study concluded that change effort increases when a method has clones. In another study [160] a resilient approach to track clone instances over time is presented. Extension, persistence and stability in methods were the chosen metrics to assess the cloning imprint and impact of clones on changeability of the application. In sample subject systems, the study concluded that cloning extension remains stable in 10–20% of the application. Otherwise cloning presents low stability, high persistence and low extension.

An ethnographic study of clones in object oriented programming paradigm was carried out by Kim et al. [122]. Software developers' copy and paste programming behavior was captured with the help of a logger. The study observed that language limitations and programmers intentions promote clones. The study pointed out the use of copying and pasting in understanding and restructuring source code. Detailed evolution analysis of type-1 clones was carried out by Göde [69]. The study included 200 revisions of nine open source systems. He found that the lifetime of clones decreased on average. The minimum clone length and programming language had little impact on the results but the lifetime of fragments differed between the systems.

4.3.2. Code clone analysis

This topic covers studies related to refactoring of code clones based on code clone classification or detection. Many software

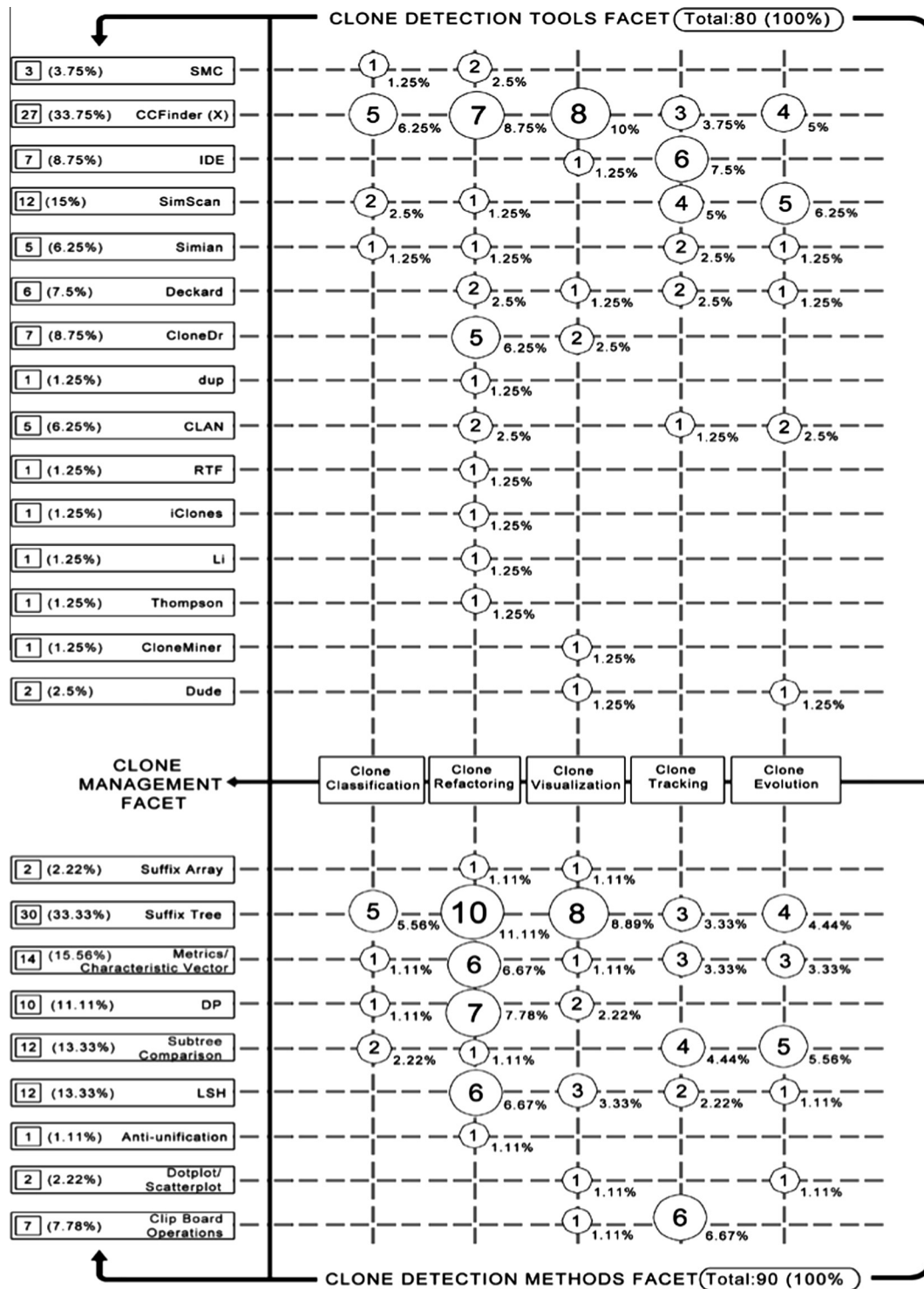


Fig. 4. Clone management map.

renovation frameworks work on improving software by miniaturization of code, i.e. clone removal. Some studies investigated cloning patterns. These are also included in this domain.

Fanta and Rajlich [60] used *CloneDR*, *dup*, and *CLAN* to develop a re-engineering process intended to eliminate clones from a proprietary software of the Ford Motor Company, i.e. Powertrain Engineering Tool sized at 120 KLOC. The study pointed out disadvantages of clones. Balazinska et al. [9,10] developed a clone re-engineering tool *ClorT* which works with any AST based clone detector. Their analysis focused on two aspects of clones, i.e. the

meaning of their differences from programmers' point of view and context analysis which help in refactoring. The study concluded that clones are good candidates for refactoring. The clone analysis tool *Gemini* [225] takes the output of *CCFinder* and displays the results in the form of scatter plot and metrics graph. The tool provides users with useful functions for analysis, maintenance and refactoring of code.

Higo et al. [78,79,82] suggested code clone analysis based on a refactoring perspective. The tool *ARIES* gives indicators for certain refactoring methods in the form of metrics from the output of

CCFinder. It helps in merging code clones. In one study [80], a tool *Libra* is developed for simultaneous modification based support method. *Libra* helps in finding clone candidates for the input file selected by the maintainer.

Kapser and Godfrey [117,118] observed that code clone detection tools produce large result sets hindering in-depth investigation of any subject system. They carried out the study on the Apache web server to gain insight into cloning patterns and understand cloning behavior. The study concluded that cloning usually occurs in related subsystems. Kapser and Godfrey [120] used *CLICS* to determine clone characteristic and location patterns. Cloning patterns are defined by what, why and how cloning takes place. In the sample subject system, the study concluded that patterns have good impact on a software system if used carefully. Quo et al. [183] stressed the need to shift the onus of pattern mining of clones from spatial space analysis to logic domain analysis. The authors use a PDG as the source code representation and propose a joint space-logic domain framework for pattern mining. The proposed approach mines two lists of patterns: one ordered by reused times and other by size of patterns. Former patterns provide good inputs for code optimization. The tool helps in locating related defects across identical pieces of code.

Tairas et al. [212,217,218] developed *CeDAR* (clone detection, analysis and refactoring) which takes the output from *CloneDR* and other clone detection tools. It helps in refactoring of detected code clones. Tairas and Gray [219] applied latent semantic indexing to find relationships in clone classes, thereby helping in clustering and maintenance of clones. Clone detection was done with the help of *CCFinder* on a sample subject systems viz. MS Windows NT source code kernel. In one study, Tairas et al. [221] used *CeDAR* to represent clone groups in a localized manner and information about each clone in a group could be viewed in one location. Each clone in the group is represented as an abstract syntax tree, after then suffix tree is used to trace similarities and differences. The technique is capable of finding near miss clones and uses *Deckard* as a back end. *Deckard* was also used for sub-clone refactoring [220] in open source software artifacts.

Jarzabek and Li [97] found out that 68% of code in Java Buffer library, JDK1.5 was contained in cloned classes or class methods. Manual investigation of situations leading to clones revealed difficulties in their elimination. The authors proposed a generative programming technique of XML based variant configuration language (*XVCL*) to analyze and represent clones. Tiarks et al. [222] found a large number of false positives, i.e. as high as 75%. This has an adverse impact of clone understandability. It also hinders the use of clone detection for practical purposes. Respective studies were carried out to detect and remove code clones from Erlang/OTP [153] and Haskell [27] programs. Jacob et al. [95] used different mechanisms to compute differences in clones. Their study used metrics and the Levenshtein distance to display the changes interactively in the code to the developer.

Juergens and Göde [107] successfully used clone coupling to detect relevant clones from a large number of false positives. The clone detector is re-run to improve accuracy by removing false positives. Shawky and Ali [205] assessed a set of metrics for similarity prediction for clone detection. Precision and recall were calculated for every experiment. They concluded that the order in which metrics are used for clone detection affects results.

Krinke et al. [140,141] analyzed the version control system to distinguish the original from the copied code. Each clone pair is classified as identical, copied or unclassifiable. The clones of a clone pair are said to be classifiable with a tolerance based on the Levenshtein distance. In a particular case study [141] for GNOME projects, more than 60% of the clone pairs could be separated into original and copy. Increasing the minimal clone size in the clone detection tool, i.e. *Simian*, the number of clone pairs decreased

asymptotically. Choi et al. [38] used a combination of clone metrics to extract code clones from the source code ready for refactoring. The metric graph of *Gemini* was used to analyze the output of *CCFinder* using metrics namely: average length of token sequence in a clone set, ratio of non-repeated token sequences of code clones in a clone set, number of code clones in a clone set. Basit et al. [15] analysed patterns of clones in STL using *CCFinder*. The study reports a high rate of cloning in container classes and significantly less in algorithms of STL. The authors proposed an XML-based variant configuration languages based clone free representation for STL's under study. In another work, Basit et al. [18] carried out a study to analyze the presence of simple clones in structural clones. They used the clone detection tool, *Clone Miner*. Across a comprehensive set of 11 subject systems, structural clones are analyzed to identify their location, similarities, etc. The analysis of structural clones, i.e. high level similarities is intended to help in understanding, refactoring and maintenance of the software system. The study concluded that over 50% simple clones were a part of structural clones.

4.3.3. Impact of software clones on software quality

Do clones have adverse effect on system quality, i.e. maintainability, reliability, etc.? This has been a matter of extensive discussion in research community. This section summarizes the related studies on whether the clones are harmful or not in the form of a table. In this section, we also include studies concerning detection of bugs as side effect of cloning and inconsistent modification.

Monden et al. [172] used *ARIES* to discover the relation between software clones and software quality attributes like reliability and maintainability. The study concluded that modules (files) having large code clones (more than 200 LOC) are less reliable and maintainable than non-clone modules. Imai et al. [91] estimated the maintenance cost caused by clones. The study measured functional redundancy which is a degree of propagation of clone potential function. After clustering, a functional redundancy tree is constructed where the weight given to each node of the tree indicates cost.

Krinke [142] carried out an important study to compare the stability of cloned code and non-cloned code. He studied changes particularly deletions, additions and changes of five open source systems across 200 weeks in cloned code and non-cloned code. This study was extended by Göde and Harder [71] using different parameters and detailed measurements. They concluded that clone detection parameters influence the results but do not change the relation between stability and non-stability. They also concluded that type-1 clones are less stable than type-2 and type-3 clones. In another study [72], they analyzed patterns of consecutive changes to clones and their impact on unwanted inconsistencies. The study concluded that consecutive changes are not good choice to detect unwanted inconsistencies. Moreover detected unwanted inconsistencies had very low severity. In addition to [142,71], Krinke [143] investigated the age of code as a parameter for clone stability. He calculated the average age of cloned code and concluded that cloned code in a file is usually older than non-cloned code, thus more stable.

Table 11 lists the related studies on whether the clones are harmful or not. In addition to the studies which analysed the relation between code clones and software quality to know the impact of clones, we also listed some studies like Kapser and Godfrey [120] which categorized the cloning patterns to be harmful or harmless. Some of the studies mentioned in the table are discussed in other sections of the paper.

It is always problematic to identify consistently changing clones over time. Thus a clone tracking and awareness tool is essential to assist software developers in efficient maintenance of software. Jablonski and Hou [94] identified features of *CnP* which help in cutting software maintenance costs. Clone information during

Table 11

Studies related to whether the clones are harmful or not.

Sr. no.	Study	Focus of the study	Findings/results
1	Monden et al. [172]	Relation between code clones and software reliability and maintainability	Clone included modules are more reliable and less maintainable than non-cloned modules
2	Kim et al. [122]	Analysis of programmers' copy and paste programming practices	Cloning is not always harmful as many of the clones are intentionally introduced by the programmer which helps in fast development and program understanding
3	Lozano et al. [158]	Analysis of harmfulness of cloning based upon change based experiment at method level granularity	Clones are harmful and have bad impact on maintenance
4	Kapser and Godfrey [120]	Categorization of cloning patterns in programs to know whether they are useful or not	Cloning is not always harmful and some of the cloning patterns are beneficial to software development and maintenance
5	Krinke [142]	Analysis of stability in terms of changes to the system from an analysis of 200 weeks of evolution	Cloned code is more stable than non-cloned code
6	Juergens et al. [106]	Analysis of inconsistent changes to code clones and whether these changes lead to defects	Inconsistent clones are a major source of faults thus increasing maintenance
7	Juergens et al. [108]	Clone detection is done in requirement specification documents by customizing code clone detection tool	Cloning is harmful in requirement specification documents
8	Rahman et al. [184]	Analysis of relationship between cloning and defect proneness	Cloning do not introduce more bugs thus not harmful
9	Selim et al. [204]	Whether cloning is harmful? What features of cloned code make it error prone?	Harmfulness of cloning is subject system dependent.
10	Bettenburg et al. [21]	Analysis of effect of inconsistent changes to code clones on software quality at release level	Cloning do not effect post release quality of the subject system
11	Göde and Harder [71]	Analysis of stability of the software system in terms of deletions to the cloned code and non-cloned code	Cloned code is more stable than non-cloned code
12	Krinke [143]	Analysis of stability of the software system using average age of cloned code in comparison to non-cloned code	Cloned code is more stable than non-cloned code

software development helps programmers in modification and debugging tasks. Visualization and renaming features, i.e. *CReN* and *LexID* of *CnP* were tested thoroughly. The study showed the effect of clone information on maintenance. Bettenburg et al. [21] explored the effects of inconsistent changes to code clones on software quality at release level through an empirical study. Developers usually carry numerous changes in different parts of software systems across releases. They are usually interested in source code at release level.

Earlier studies investigated the impact of code clones at much finer granularity level. Duala-Ekoko et al. used *Clone Region Descriptor* [53] approach to track code clones across releases. All clone genealogies found were manually inspected for inconsistent change. Juergens et al. [108] carried out clone detection to find copy and paste activities in requirements specifications documents. Large documents are used to detect any redundancy, thereby assessing the quality of specifications. The comprehensive study was carried out across 28 documents with 9000 pages, fixing clone length to be 20 words. Rahman et al. [184] carried out an empirical study to investigate the impact of code clones on defects in code. The clones were not found to be particularly error-prone. Clone detection tool, *Deckard* and data mining from version control and bug repositories were used to carry out the study. Göde [70] found that most of the clones detected by state of the art clone detectors need not be removed. His study has significant implications from point of view of software maintenance. With standard subject systems as input and an incremental clone detector tool, the study identified procedure extraction as the most commonly used refactoring method. Kozlov et al. [137] explored internal quality attributes and code clone detection metrics. For all 117 releases of peer to peer open source software namely, eMule, a software project fork, internal quality attributes were measured using *SoftCalc* tool and code clone detection metrics were extracted using *CCFinderX*. Statistically significant correlations across groups were identified based on the Pearson product moment correlation. The study was based on a number of hypotheses taken from prominent clone detection studies using metrics. The study concluded that internal quality attributes act as explanatory variables for code clone detection metrics. Selim et al. [204] used survival models to study the

impact of clones on software defects. The probability of occurrence of a defect at any time was modeled using Cox's proportional hazard function. The study used a set of predictors to classify clones as helpful or harmful. The study was the first of its kind to use Spearman correlation between actual and predicted occurrences of defects. It explored a new set of predictors related to code siblings at method revision level. They concluded that cloned code is not always more risky than non-cloned code.

Copied code leads to inconsistencies at multiple places when a bug is introduced in the original code fragment. Automatic detection of these bugs has turned to be an allied area effecting the quality and maintainability of source code. Li et al. [152] pioneered defect detection in clones using *CP-Miner*. The technique was able to detect code duplications and related bugs using a frequent subsequence mining technique. However, the tool reports many false positives. Jiang et al. [99] stated that inconsistencies emerge when code is copied to a new place and appropriate changes are not made with reference to new context. They used *Deckard* to detect context based clone related inconsistencies (bugs). The approach was able to discover previously unknown bugs and the results were compared with *CP-Miner*. Hayase et al. [76] introduced a filtering technique to reduce the false positives generated by *CP-Miner*. *CCFinder* was used to find identifier naming inconsistencies and *CP-Miner* was used to filter code clone related bugs. Yoshida et al. [233] used lexical analysis and identifier similarities to detect duplicate code. The technique helps in detecting code fragments with similar defects. In a similar study, Yoshida et al. [235] detected similar defects in source code based on comparison of synonymous identifiers using the Jensen–Shannon divergence method. Clustering of identifiers was based on distance between identifiers. They also compared their results with *CCFinder* on the same subject systems. Juergens et al. [106] introduced the *CloneDetective* framework to detect inconsistencies. The study pointed out that inconsistent clones lead to faults. In the sample subject system, 58% of the clones contain inconsistencies. Gabel et al. [64] introduced *DejaVu*, a tool to detect inconsistent bugs. Firstly similar code fragments are found using a *Deckard* based clone detection framework. Later on, buggy change analysis is done to classify benign and buggy inconsistencies from the clone detection data set.

The potential bugs are classified as: bugs, code smells, style smells, unknown, false reports. Jalbert and Bradbury [96] used clone detection and rule evaluation to identify bugs in concurrent software. They tried to reduce the domain of testing by using clone detection. An identified bug is input to find similar bug patterns in concurrent software.

4.3.4. Clone detection in websites

Websites, i.e. web applications are usually multilingual and suffer from short development life cycles. Changing requirements frequently leads to introduction of clones. Many studies confirm the presence of clones in websites. The extent of cloned code varies from 30% [211] to as much as 63% [185].

Aversano et al. [5] proposed reuse of existing sites by means of cloning and adaptations from a repository of conceptual views and code components. Lucca et al. [161] presented an approach to detect duplicate web pages using similarity metrics. They proposed refining the results of similarity metrics using the Levenshtein distance between each pair of analyzed HTML strings. Lanubile and Mallardo [146] conducted a study to detect script function clones using metrics. The study was conducted on three sample subject systems and found 39–50% of the total script functions to be clones. Synytskyy et al. [211] and Cordy et al. [41] used island grammar for simultaneous parsing of multilingual web applications. They used UNIX *diff* command to compare potential clones. The method is able to detect near miss clones. Lucia et al. [162,163] proposed a method to identify cloning patterns in a web application. The method was based on clone detection using similarity thresholds. The technique was helpful in clone analysis and reducing the code size and navigational patterns of a web application.

Rajapakse and Jarzabek [185] used *CCFinder* to detect patterns of clones in websites. Authors extended the study further [186] to unify most of the clones using server pages. This helped in reducing code size and the possibility of update anomalies. Different clustering algorithms and latent semantic indexing were used by Lucia et al. [164] to detect clones in web applications. The techniques were tested on different static web sites. Different clustering algorithms produced comparable results. Jung et al. [111] explored three levels of views namely, relationships between web applications, passed parameters and target applications for detecting clone pairs in a web application. Clone pair candidates were selected based on static and dynamic approaches. The combined approach was successfully validated on two open source projects. Martin and Cordy [169] introduced the concept of contextual clones, i.e. clones that can only be found by augmenting code fragments with related information referenced by the fragment to give its context. They proposed a technique to leverage the idea of contextual clones to detect similarities in web service description language.

4.3.5. Cloning in related areas

Duplication is also prevalent among other software artifacts like requirement specifications and binary executables. Such studies are included in this category. Studies of different applications of clone detection are included in this category too.

Software archives contain large amount of similar software. Kawaguchi et al. [121] applied a code clone based similarity metric, a decision tree based approach and latent semantic analysis based approach to categorize similar software in an archive. The technique helps to identify relationships among software systems. Gallagher and Layman [65] found similar decomposition slices, i.e. slice clones. A decomposition slice is a program slice which captures all relevant computations from a given variable. The study presented the negative and positive points to identify slice clones as software clones.

Domain analysis is usually carried out in device drivers to search for similar implementations. Ma and Woo [165] used *CCFinder* to detect clones within the file and across files in device driver source files. Mao et al. [167] used table recognition technology and clone detection to convert conventional table-based websites to modern cascading style sheet websites. Software clone detection was used to detect common layout styles in pages. German et al. [66] studied technical and legal implications of cloning in code siblings when there is license compatibility between copyright owner and destination. Monden et al. [173] chose metrics with a lower bound on code clone measurement threshold to determine violations in open source licensing. Fifty open source subject systems were chosen from free software directory to trace violations. Brixtel et al. [26] used clone detection tool for checking plagiarism in student projects and assignments. They tested their technique on projects using different languages. Davis and Godfrey [45] used assembly instructions to detect source code clones.

Software product lines have a core part around which its different components are built. Dalgarno [43] applied clone detection in a product line context. Mende et al. [171] applied token based clone detection and the Levenshtein distance to identify similar functions which make the core part in software product lines. Schulze et al. [201] identified clones in feature oriented software product lines. Using *CCFinder*, a significant number of clones were detected in 10 subject systems. They discussed reasons for cloning in feature oriented software product lines and the way most of the clones can be refactored. Domann et al. [52] used the *CloneDetective* [105] framework to find instances of clones in 11 software requirement specification documents of 2500 pages. Clones in binary executables were detected by Saebjornsen et al. [196]. This tree based approach worked by clustering of characteristic vectors on labeled trees. Ciancarini and Favini [39] carried out a study to detect clones in game playing software. They found a criterion to judge similarities in game playing software with chess as an example. Juergens et al. [108] investigated the amount and nature of duplicated text in software requirement specifications using *CloneDetective* tool. After detecting duplications in 28 software requirement specification (SRS) documents of 8667 pages, recommendations were given for working with SRS in practice. Whaley and Lam [229] applied cloning in pointer alias analysis by creating clones of methods. They used binary decision diagrams to achieve context sensitive results.

4.3.6. Software clone detection in aspect oriented programming/cross-cutting concerns

Aspect-oriented programming (AOP) was a response to the problem of cloned code relating to cross-cutting concerns (e.g. logging, and error-handling) in object-oriented systems. The detection of code that should be refactored into an aspect is an important part of AOP.

Yokomori et al. [232] analyzed the relationships between aspects to understand how the behavior of existing code clones in original classes spread to aspects. The study used *CCFinder* to detect code clones. The study concluded that number of clone relations between class and aspect increases if only one part of clone group is extracted. Bruntink [28] used clone metrics to filter clone detection results to detect aspect candidates. A grade was associated with each clone class to rank its suitability in improving maintenance. Bruntink et al. [29] conducted a study where the token-based *CCFinder*, AST-based *ccdimpl* and PDG-based *PDG-DUP* were evaluated in terms of finding cross-cutting concerns in C programs with homogeneous implementations. Some well known cross-cutting concerns such as error handling, tracing, range checking, null-value checking and memory error handling were found. Their study showed that both *ccdimpl* and *CCFinder* are best suited for null-value checking and error handling concerns while *ccdimpl* is

also suitable for the range checking concern. *PDG-DUP* can find tracing and memory handling concerns. The study confirmed that code clones contribute 25% of the code size in aspects. Schulze et al. [200] used code clone classification to decide whether to use object-oriented refactoring or aspect-oriented refactoring. Code clone classification was done by adding semantic information to clone detection tool.

Fig. 3 shows the number of publications in different key areas in the years 1997–2011. Different trends can be seen for different key areas. Research in the area of impact of clones on software quality rose most dramatically in 2010 from a single study in 2009 to ten studies in 2010. The number of publications in area of code clone analysis grew sharply, almost doubling from about one study in 2007, two studies in 2008, five studies in 2009 and eight studies in 2010. The research in the key areas of code clone analysis peaked in 2010. It shows the improvement in recognition and need of research in these areas recently. On the contrary, research in area of clone detection in AOP comprised the smallest numbers. On the other hand, number of publications in the key areas of clone detection in websites, cloning in related areas and code clone evolution remained stable throughout the years.

4.4. Current status of clone management

Clone management is a set of activities like clone classification, refactoring, visualization, tracking and evolution. It plays a pivotal role in development and maintenance process. Some of the benefits of clone management are discussed in Section 4.4.1. However, it also overlaps with clone analysis, clone evolution, and impact of clones on software quality which are discussed in Section 4.3.

4.4.1. Benefits of clone management

In literature, we came across many studies which discussed benefits of clone management. Some of the findings are:

- Laguë et al. [144] identified that an effective clone management strategy improves customer satisfaction and software system quality.
- Kim et al. [122] pointed out that many clones are intentionally introduced in code due to language restrictions. Thus refactoring after clone detection may not improve the software. So it is more important to manage the clones to see how they evolve over time. Moreover, clone management in an Integrated Development (IDE) makes developers concerned about duplication. The developer should be informed in the IDE about all the clones which are introduced deliberately due to hard time constraints, etc.
- Kapser and Godfrey [120] identified cloning patterns which are helpful in improving the quality of the system. They found that as many as 71% clones had a positive impact on the maintainability of the software. They stressed the importance of managing code clones using synchronous maintenance of code clones.
- Duala-Ekoko and Robillard [53] proposed managing code clones by notifying developers of modifications to clone regions. They developed a clone tracking system which works as the system evolves.
- Jablonski and Hou [94] concluded that clone awareness with visualization and consistent identifier renaming support help developers during debugging and modifications. The study highlights the importance of clone management.

We highlight the importance of clone management by showing the percentage of cloned code from selected studies. Studies are selected to show different clone detection techniques and subject systems. It reflects the percentage by which the code can be shrunk in software (see Table 12).

4.4.2. Clone management – a cross cutting and an umbrella activity

Code clone management is an umbrella activity covering all aspects regarding clones. It is a superset including clone analysis, clone evolution, code clone taxonomies, code clone classification, etc. It also covers code clone visualization. We discussed here those studies which focused on clone management and clone visualization. Moreover, the relevant information is spread across many themes addressing research question 3 in Section 4.3.

Balazinska et al. [8] focused on restructuring and reengineering software after successful clone detection. Using six open source subject systems written in Java, method clones were manually analyzed for reengineering opportunities. Higo et al. [77] provided a technique to identify meaningful blocks in code clones that are easy to merge. They used *CCFinder* to detect code clones. The technique is helpful in reducing number of clone pairs detected in two open source Java systems. Kapser and Godfrey [116] investigated two large subject systems to classify code clones, viz. function clones and partial function clones. The authors suggested management of code clones by classification and filtering false positives. *CRen* [93] developed by Jablonski and Hou provides programmers with identifier renaming support and tracking of clones in an integrated development environment. The authors extended their work in the form of *CnP* [92]. *CnP* [86,87] is intended to support and manage clones proactively as they are created and evolved. The tool was developed as Eclipse plug-in.

Tairas [216] proposed a technique to unify clone detection, analysis and refactoring. The study presented a method to improve clone maintenance by eliminating redundant code by identifying refactoring opportunities. Nguyen et al. proposed *Cleman* [174], a framework for comprehensive code clone group management in evolving software. They developed *Clever* [175], a clone aware software configuration management system which works with any AST based clone detection tool. It performs umbrella activities like clone detection, clone change management, clone consistency validating, and clone merging. Duala-Ekoko and Robillard developed *CloneTracker* [54] to track clones during the evolution of source code. It uses *SimScan* as clone detection tool. The authors used *CloneTracker* to produce clone region descriptors (CRDs) [55] which are an abstract combination of lexical, syntactical and structural information. Clones are tracked using CRD which represents a clone region for different clone groups which are of interest to a developer. This approach goes beyond code based clone descriptors in integrated development environments like *CRen*. *CloneBoard* [46] is a clone management tool which infers clone relations dynamically by monitoring clipboard activity. The tool is integrated in Eclipse to support copy pasted clones. Lee et al. [151] introduced a scalable and instant code clone search technique for use during software development. The technique works by extracting characteristic vectors from the source code. Then a multidimensional indexing tree structure *R*tree* is used. In the index, filtering and ranking is used to evaluate code clone detection queries in order. The authors also devised an approximate clone detection technique which is fast but less accurate. Both the algorithms gave sub second response time for processing a million lines of source code.

4.4.2.1. Clone visualization. Code clone visualization is one of the most important areas of code clone management after successful code clone detection. With the increase in duplication in various software artifacts, e.g. source code, proper representation of software clones has become a challenge. This domain covers presentation of duplicated code which helps in fast analysis of clone detection results.

Several clone detection tools report the presence of clones in form of starting and ending line number, file name, etc. One of the widely accepted formats for representation of clones is scatter

Table 12

Percentage of cloned code across different systems.

Sr. no.	Percentage of cloned code (%)	Subject system	Size (LOC)	Clone detection technique	Citation
1	13–20	SS subsystem	1.1 M	Token based	[7]
2	5–20	Telecommunication monitoring systems	1 M	Metrics based	[170]
3	12.7	Process control system	400 K	Tree based	[22]
4	50	GCC and other application projects.	6.5–460 K	Text based	[56]
5	8.3–14.8	E-business website and resource management system	15–35 Sequence diagrams	Model based	[155]
6	20–50	Java source code, C# source code	425 K, 16 K	Tree based	[58]
7	14.9	SRS	2500 pages	Text based	[52]
8	12.1–32.1	Open source python systems	9–272 KLOC	Hybrid	[194]

plot. Tairas et al. [213,215] extended the *AspectJ Development Tool* visualizer to display the results of *CloneDR* in Eclipse framework. The main application of the tool is to display the cross cutting concerns in aspect oriented programs. Adar and Kim [1] developed *SoftGUESS* which supports exploration and visualization of code clones. Their system supported different views of analysis of clones over single and multiple versions, through analysis of graphs. Jiang and Hassan [101] developed a *Clone System Hierarchical Graph*, an interactive graph used to select nodes to highlight how the clones are scattered in a particular directory. They used clone mining to highlight clones at different levels of abstraction.

Most of the clone visualization tools like *Gemini* [81] read the output of *CCFinder*. These tools filter uninteresting code clones and navigate to clones having features the users are interested in. Zhang et al. [236] developed a standardized graphical representation as an Eclipse plug-in called *Clone Visualizer* to filter and visualize the output of *Clone Miner*. Fukushima et al. [62] used a code clone graph to visualize the output of *CCFinder*. Nodes of a code clone graph correspond to code clone sets and edges represent clone sets in the same file. The technique helps in representing diffused clones in which clone set clusters are located in files having different functionality. Tairas [217] presented a technique in which one clone instance displays the properties of all the clones in the clone group. The technique was integrated as an Eclipse plug-in called *CeDAR* (Clone Detection, Analysis and Refactoring). This representation help in refactoring as clone group representation displays the difference among clone instances.

4.4.3. Clone management: a systematic map

Clone detection tools are applied to detect clones after the software development is completed in a postmortem approach of clone management. But even the leading clone detection tools report a large number of false positives. So recently, researchers and practitioners have tried to make clone detection an integrated part of development environment to increase the effectiveness of clone management. Developers need to be informed online as and when clones proliferate.

Clone management is a cross cutting topic touching different domains of software clones. We identified four key areas in this systematic literature review, i.e. clone analysis, clone evolution, impact of clones on software quality and clone visualization which touch clone management. The first of these three key areas has been discussed in detail in the context of research question 3. The papers from these key areas were studied to identify relevant sub topics of clone management facet below:

1. Code clone classification.
2. Code clone refactoring.
3. Code clone visualization.
4. Code clone tracking.
5. Code clone evolution.

We used the systematic mapping method of Petersen et al. [181] to map clone management papers with clone detection

method papers and clone detection tool papers. From the set of primary studies, we identified 49 relevant papers. Papers were classified based on these three different facets and the results are presented in the form of bubble plot as shown in Fig. 4.

The bubbles show the number of publications identified for each clone management facet with clone detection method facet and clone detection tool facet. Total number of papers on either side of the map is not equal as many clone detection tools use more than one method for clone detection. Our bubble plot shows that the majority of research is in clone refactoring using the suffix tree and dynamic programming methods of clone detection. The metrics/characteristic vectors method of clone detection is frequently used in different sub topics of clone management. We observed the use of clip board operations as clone detection method in clone tracking and visualization. This is due to the fact that clip board activity captures in the initial creation of a clone in an IDE. *CCFinder* (X) is the most frequently cited tool. It has been used for clone classification, clone refactoring and clone visualization as it is able to detect large number of clone candidates with high recall. The map suggests a lack of research in clone classification. Barring *CCFinder*(X), *SimScan*, *CloneDR*, *Decard*, *Simian*, and *CLAN*, the rest of the tools are only used in one or two papers.

4.5. Subject systems

We have observed that different subject systems are being used in clone detection research. We are hopeful that the above table may help researchers in choosing most commonly used subject system as benchmark for evaluation and empirical studies. We list 28 open source subject systems that were subjects of different studies. Table 13 lists all open source software systems and Table 14 lists commercial systems. We also list approximate size in LOC, programming language of the system and usage count. The usage of the subject system according to our classification of literature and citations is also mentioned in the table.

Recently clones in Matlab/Simulink models are detected by [47,49,88,182]. Studies [48,49,88] have used common models (SIM,MUL,SEM,ECW) available from Matlab central. The size of the model in blocks varied from 440–18,000 blocks. Domann et al. [52] and Juergens et al. [108] applied clone detection to 11 and 28 requirement specifications respectively from a total of 2500–8667 pages.

We have omitted those open source subject systems from Table 13 which are only used in one study. Different versions of the same program are mentioned in one place and the size of latest version used in any study is included. For instance, different versions of Linux have been used by [16,113,152,193], etc. These are written in the one place and the largest size among them is reported.

JDK has been used many times for clone detection and clone analysis. The Apache web server and Linux kernel and its different versions have been used extensively as subject systems to carry out the clone detection and analysis study. Similarly, the software

Table 13

Open source subject systems.

Sr. no.	Subject system	Size (LOC)	Language	#	Classification	Citations
1	JDK	3200 K	Java	10	Clone detection Clone analysis Comparison and evaluation	[36,113,151,176,179,202,226] [8,97] [202]
2	Apache-httpd	343 K	C	9	Clone detection Clone analysis	[68,125,150,152,193,194] [120,140,184]
3	Apache-Ant	1.41 M	Java	4	Clone analysis Impact of software clones	[18,82,221] [204]
4	ArgoUML	1.76 M	Java	10	Clone evolution Clone analysis Impact of software clones Clone detection	[6] [140,221] [21,70,71,142,143,204] [68]
5	Linux	6.2 M	C	10	Clone detection Clone analysis Impact of software clones	[16,34,74,89,113,152,193,194] [101] [233]
6	Weltab	11 K	C	7	Clone detection Clone analysis	[57,129,180,193,194,214] [205]
7	Netbeans-javadoc	19 K	Java	7	Clone detection Comparison and evaluation	[13,59,84,193,194,203] [58]
8	jEdit	157 K	Java	5	Clone detection Clone analysis Impact of software clones	[24] [18,221] [21,143]
9	Bison	16 K	C	5	Clone detection	[59,130,134,138,193]
10	PostgreSQL	937 K	C	6	Clone detection	[59,125,134,152,193,194]
11	Snns	105 K	C	5	Clone detection	[59,134,180,193,194]
12	FreeBSD	403 M	C	3	Clone detection Clone analysis	[113,152] [157]
13	ANTLR	61 K	Java	3	Clone detection Clone analysis	[179] [8,18]
14	Eclipse-Ant	35 K	Java	7	Comparison and evaluation Clone detection	[58,193,194] [13,59,68,84]
15	Wget	17 K	C	3	Clone detection	[59,193,134]
16	Cook	70 K	C	3	Clone detection	[57,180,193]
17	Abyss	1500 K	C	3	Clone detection Clone analysis	[193,214] [205]
18	Tomcat	130 k 167 K	Java	3	Comparison and analysis Clone detection	[148] [24,147]
19	SquirrelL	218 K	Java	3	Clone analysis Impact of software clones	[221] [71,142]
20	GCC	1.2 M	C	3	Clone detection	[56,68,104]
21	FileZilla	90 K	C++	3	Impact of software clones Clone evolution Impact of software clones	[142] [206] [108]
22	Tcsh	45 K	C	2	Clone detection Clone analysis	[132] [133]
23	Bash	40 K	C	2	Clone detection Clone analysis	[132] [133]
24	CLIPS	34 K	C	2	Clone detection Clone analysis	[132] [133]
25	Jabref	114 K	Java	2	Clone detection	[89,109]
26	DNSJava	25 K	Java	2	Clone evolution Clone analysis	[6] [18]
27	Eclipse-jdtcore	148 K	Java	8	Comparison and evaluation Clone detection Impact of software clones	[58] [13,59,84,95,193,194] [142]
28	j2sdk1.4.0-javawswing	204 K	Java	7	Comparison and evaluation Clone detection Clone analysis	[58] [13,59,84,193,194] [18]

system ArgoUML has been used in recent studies to investigate the impact of software clones.

Subject systems in C (weltab, snns, postgresql) and in Java (netbeans-javadoc, eclipse ant, eclipse-jdtcore and j2sdk1.4.0-javaw-

swing) used by Bellon's experiment [20] are frequently used by researchers. Most of the research is carried out using subject systems written in C and Java. This may be due to a lack of suitability of tool for other languages. The efficiency of existing tools should

Table 14
Commercial subject systems.

Sr. no.	Subject System	Size (LOC)	Language	#	Classification	Citation
1	Government system	1 M	COBOL and PL/1	1	Clone detection	[113]
2	HagerROM (Würzburg University)	87 K	Java	1	Clone detection	[226]
3	DISLOG development Kit (DDK)	95 K	PROLOG	1	Clone detection	[226]
4	SPARS-J	47 K	C	1	Clone analysis	[157]
5	Commercial	460 K	ABAP	1	Clone detection	[89]
6	Commercial CAD	1.6 M	C	1	Clone detection	[230]
7	Graph-layout program (IBM)	11 K	C	1	Clone detection	[130]

be measured using other languages which the tools support. Research up to the end of 2011 have predominantly been done on open source systems. Few commercial systems have been used. Clearly the use of open source systems is preferable from the viewpoint of repeatability of experiments and also allows tool comparisons to be easily extended to cover new or amended tools.

5. Discussion

We surveyed 213 articles out of a collection of 2039 and provided categorization and quantitative overview. Unlike previous surveys, we put an emphasis on clone management, model clones and semantic clones and classified the literature from different key areas. Existing surveys/technical reports by Roy and Cordy [187] and Koschke [135] consider research findings till 2007. These surveys filled the initial void for useful text for budding researchers in this domain. The work by Roy et al. [192] focused on clone detection tools and techniques up to 2009. Pate et al. [178] presented systematic review of existing literature on clone evolution. The authors framed three research questions to investigate what methods have been used to study clone evolution, to study cloning patterns and to discover the presence of consistent/inconsistent changes to clones during evolution. The authors identified 30 primary studies regarding clone evolution. Our focus is broader than the earlier surveys and includes the latest research work related to software clones up to mid 2011 using the systematic literature review guidelines of Kitchenham and Charters [127]. In addition to clone detection tools and methods, we have addressed other issues related to software clone research such as clone analysis, clone evolution, and impact of clones on software quality. We used a systematic method to develop a clone management map which identifies how clone management papers overlap with clone detection method papers and clone detection tool papers. We explored the model based and semantic clones in detail and compared the state of the art techniques. Moreover, major breakthroughs in model clone detection happened after 2007. We presented all the studies in different sections in chronological order which makes it easy to identify the latest research carried out after 2007 as done in earlier surveys.

This section discusses the principal findings of our systematic review, strengths and weaknesses of the evidence. We begin with the discussion of key sub areas followed by clone management. Implications for researchers and practitioners are summarized after that, followed by limitations of the review.

5.1. Key sub areas

We divided the literature into six different key areas realizing the importance from research perspective. We noticed that some of the areas are inter-related.

It is not an easy task to model clone evolution under a number of versions. The prediction accuracy of the approach depends upon parameters and their variation. Future research should consider using sound mathematical modeling approaches. Many clone

genealogies are alive and long lived and clones are easy to manage in smaller systems as compared to large systems. The area is still open for research to study clone genealogies using state-of-the-art clone detection tools in different sample subject systems.

A large numbers of studies confirm the harmfulness of cloning in software systems. Less research was found in tracing useful patterns of cloning which help the programmer. We observed limited number of studies investigating cloning patterns in different programming paradigms. More studies should be carried out to investigate the types of clones and their characteristics like persistence over time in different programming methodologies. In the initial years, we found few studies attempting to calculate the impact of code clones on maintenance cost. We noticed increase in the number of studies undertaken to compare the behavior of cloned and non-cloned code and their impact on system quality during last 2 years. Most of these studies use version control information. However, these systems detect minor changes like white spaces too, which should be ignored from the clones point of view. We found contradictions among the papers and the area is still open, since the number of subject systems is too low to arrive at any general conclusion. Future research lies in carrying out the same study with large number of comparison parameters, clone detection tools and subject systems. Recently, many studies presented findings that clones in general do not have adverse effect on quality. At the same time, we observed some contradicting studies. Such studies need to be extended using external quality attributes and a large subject base. The nature of the subject system and programmers' behavior has a profound effect on these studies. In one study, varied results were noted for the ArgoUML and Ant subject systems. We need more studies to accurately calculate the increase in maintenance cost of software due to presence of different types of clones.

We found two studies that checked the presence of clones in software requirement specifications. There is a need to conduct clone detection studies to investigate redundancy throughout all documents of software development life cycle. Such repetitions when removed will lower the maintenance cost of the software in earlier phases of software development life cycle. We found only one study which applied clone detection to trace open source licensing violations. Such studies will be helpful to distinguish between reuse based and accidentally produced clones.

5.2. Clone management – a cross cutting topic

Different clone management tools should be evaluated depending upon use cases and future tools can be developed catering to respective use cases.

The systematic map in Fig. 3 helps in identifying which sub topics of clone management have been emphasized, which areas require further research, which clone detection tools and methods have high usage in clone management. In the map, we did not find any papers involving more than one tool in any aspect of clone management. This may indicate a lack of interoperability between different tools for clone management. We find a lack of work in clone classification. To assist the software maintainer, it is impor-

tant to classify clones as “clones to be retained” and “clones to be removed”. We found large number of studies regarding maintenance of code clones by identifying refactoring opportunities. There is dearth of studies validating the analytical results of clone detection, clone analysis, clone management, etc. with developers intent and behavior.

5.3. Implications for research and practice

The systematic review has implications for researchers who are looking for new concepts in the field of software clones, and for practitioners working in software companies who want to apply clone detection for software development.

As the research community is still arguing on the exact definition of the term clone, it is imperative to devise automatic oracles for all types of clones. Studies confirm that there is disagreement between human experts as to whether the candidate code is or is not a clone. It is a difficult and time-consuming task to manually classify the candidates as clones or not. Thus, we believe that experts from industry and academia related to diverse domains of clone detection should come together to create a verified reference corpus of clone candidates in standard subject systems. The study should be carried out differently for each type of clone and depending upon use case. Such benchmark suites would make the results of empirical comparison consistent and reliable for use in research and industry.

For researcher and practitioners, a number of avenues are open. Clone management has emerged as a challenging area. Software developers in industry deal with large amounts of data. So clone management tools should be scalable and integrated in development environments, to help programmers understand the behavior of cloning patterns. A comprehensive industrial strength clone management tool having integrated detection and developer friendly visualization of clones would help the developers observe clones as and when they proliferate during development.

The application of clone detection tool depends upon situations and objectives. There are different circumstances where clone detection is essential. Some of the areas are: aspect mining [28], to find cross cutting code [29], plagiarism detection [26], software product lines [171,201], clones in web sites [146,185], origin analysis [159], quality assessment [172], detecting licensing violations [66]. Though these areas are independent research fields, yet these areas and clone detection can get benefited from each other. Usually cross cutting code is scattered in different implementations of the program. These implementations tend to be functionally similar, so semantic clone detection technique is helpful in finding cross cutting code. In plagiarism detection, the code is copied and disguised intentionally. By representing the code in an abstract representation like PDG, existing code clone detection tools may be customized to detect hidden changes in the code. Clone detection helps in detecting shared and common set of features in software product lines. Existing systems can be reengineered to obtain reusable assets with the help of clone detection. Origin analysis is the study of detecting the location of changes to the system from one version to the next. Clone detection may help in origin analysis with detection of similar function/class/file across versions. Code clone across systems may directly lead to licensing violations and copyright infringements. So clone detection technique can be applied to detect these violations.

5.4. Limitations of this review

The main limitation of this study is multitude of meanings associated with the keyword ‘clone’. We tried to be extremely cautious in data extraction. Strings like code duplication, redundancy were manually searched in databases to increase the number of research

articles in our study. However, manual searches may miss relevant articles.

Data extraction was carried independently by the researchers. But only one author reviewed the discarded articles. As far as the classification of studies is concerned, there were disagreements among researchers. Each researcher classified all papers individually before comparing the results. In cases where there was disagreement, the issue was discussed until consensus was reached. Thus the papers were classified in several different categories.

6. Conclusions and future work

In this review paper we identified 213 studies from literature, of which 100 were found to be research studies of software clone detection. We have presented the results in different dimensions like classification of clone research, code clone management as cross cutting domain, types of clones, clone detection tools, clone detection approaches, internal representations, subject systems, semantic clones and model clones.

We noticed a great variation in the definition of the term “clones”. We realized the series of the International Workshops on Detection of Software Clones have made a significant contribution towards promotion of research in the field of software clones. There are recent studies that show that clones can be often used as principled reengineering techniques, and can be beneficial in many ways. Also, it is not easy to refactor all the clones due to cost/risk associated with refactoring. So it is suggested that instead of removing clones, we should have proper clone management facilities. In order to advance the state of the art in clone management, one needs to know the advances in clone research itself. We have attempted to do this by finding the relevant literature and summarizing it in the form of systematic map. This survey is helpful in finding the research gaps in area of software cloning in general and clone detection in particular.

It is still arguable whether clones are harmful or not. Undoubtedly in large case studies some of the clones are intentionally introduced. More studies need to be undertaken to know how harmful actually clones are. Saha et al. [198] stated that it is important to understand code clone evolution to know whether clones are harmful or not and to know the impact of code clones on maintenance. Automatic tools for investigation and visualization of clone genealogies across different versions of the software are helpful for developers.

Is cloned code really defect-prone than non-cloned code? There are too many contradicting studies. The area is still open for a general statement. Different attributes of software quality coexist with software clones. The behavior and impact due to code clones is still not known. The study of changeability of cloned vs non-cloned system depends on the choice of application as some applications are continuously restructured. Such studies should be empirically carried out using different types of clone detectors on large subject system base to conclude general remarks. Kamiya [115] found out that code fragments do not appear consistently in all revisions of the software. The code fragment skips one version and again reappears in the next revision. This non-continuous code appearing across revisions of the software need to be validated with developers’ intent as to why code has been copied from the old revisions.

Godfrey et al. [73] stressed the need to track the history after code clones have been detected. It will help in analyzing the reasons as why cloning occurred in first place. Knowing the reasons as to why clones appear have implications on clone detection process in particular and clone detection research in general.

There is dearth of research in cloning beyond source code. Juer-gens [110] identified different software artifacts where cloning may occur. He emphasized that phenomenon of clone detection, reasons for its occurrence, effects of cloning has been studied

thoroughly in source code. Clones do occur in requirement specifications, models and test cases too. There is an urgent need to explore the reasons for clones and efficient clone detection techniques in these software artifacts. Different artifacts have inherent characteristics which have to be exploited to apply the clone detection algorithm for that artifact. Empirical studies need to be carried out to understand the patterns in clone evolution for these artifacts. Artifact repositories will help in benchmark and comparative studies for different artifacts. Studies regarding clone management and their economic trade off's need to be carried out for each software artifact. We realize that if clones are removed in earlier phases of software development life cycle, maintenance costs will be cut in the delivered software. We realize the research direction in use of frequent itemset mining to detect clones in textual representation of other software artifacts. Latest programming paradigms and mobile based software need to be tested for clones.

Carver et al. [33] emphasized the need to complement the analytical results of the tool with empirical behavioral studies of the developer. Human based studies can be used to validate the results of the analytical study. Various studies take static code as source code. But the behavior changes at run time. Dynamic code behavior at run time is not captured yet as part of software clone detection. The use of self learning technique and history based log techniques can be applied in the field of software clone detection.

Dang et al. [44] discussed seamless integration of clone detection tool in the IDE. The tool should be flexible, scalable and efficient in detecting real clones (clones which are really interesting and useful to the developer) and should not report non-useful and uninteresting clones. Research can be carried out to develop automatic tools to check the consistency and compatibility of licenses in code siblings. There are many language specific issues which hinder code clone classification. Subjective studies carried out by several human experts vary a lot on creating reference data for creating different benchmark suits.

The reliable and scalable detection of behaviorally similar code is an open research area. In software reuse we need to identify the most relevant component for given context. From existing software repository, developer may find many candidate components for given context, the components that are semantically same but may differ in one or other criteria such as cyclomatic complexity, algorithmic complexity, time/space trade-off, and known bugs. With the help of semantic clone detection, developer will be able to find out most suitable and efficient component out of available candidate components.

We examine that there is clear need to address the lack of empirical studies to examine the effects of cloning in real world models. Comparison and evaluation of clone detection technique in model driven development can help in choosing the right technique based on application. There is apparent need to classify the comparative and empirical studies differently for Matlab/Simulink models, UML models and data flow models as the application area is different. Storrlé [209] proposed optimal alternative algorithms for model matching for efficient clone detection of UML models. We propose to collect comprehensive real world UML models and Matlab/Simulink models and designing model clone oracle to effectively evaluate model clone detection approaches. To enhance the slow speed of model clone detection and PDG based clone detection is an open research issue. Higo et al. [80] proposed the use of inter procedural PDG in detecting identical functionalities. We foresee the use of multiple threads or parallel machines to speed up and distribute the task of retrieval and detection in future.

We hope that our study will be useful for any researcher who wants to carry forward the research in any domain pertaining to software clones such as clone management, clone detection, clone

analysis, and impact of software clones on software quality. Further our study is extended to model and semantic clone detection techniques.

Acknowledgements

We thank editor and anonymous reviewers for worthy comments. The comments and suggestions are extremely helpful in improving the research paper draft. We are also grateful to All India Council for Technical Education (AICTE), Govt. of India for funding project titled "UML based Automated Test Case Generation" for second author.

Appendix A. A quality assessment forms

A.1. Screening question

Section – 1

Does the research paper refer to software clones?	
Consider:	<input type="checkbox"/> Yes
The paper includes the study of software clones. All types of studies, i.e. case study, experimental study or research paper is included. The word 'clone' has got different meanings, so inclusion is highly cautious	<input type="checkbox"/> No

Section – 1 is evaluated first. If the reply is positive, we proceeded to section -2.

A.2. Screening question

Section – 2

Key sub-area categorization	
Is the research paper focus on software clone detection?	<input type="checkbox"/> Yes
Consider:	<input type="checkbox"/> No
– Is the study's focus or main focus on software clone detection or not?	
– Did the study fit in any one of sub-areas categorized? (Apparently the study motivated different categories.)	

If the study's primary focus is on software clone detection, proceed to section – 3, else proceed to section – 4.

A.3. Detailed questions

Section – 3

Findings	
Is there clear statement of the findings?	<input type="checkbox"/> Yes
Consider:	<input type="checkbox"/> No
Did the study mention the approach/clone detection technique?	
Has the match detection technique reported?	
What is the corresponding transformation technique,	

Appendix A.3 (continued)

Findings i.e. source representation?	
Comparison	
Was the data reported sufficient for comparative analysis?	<input type="checkbox"/> Yes
Consider:	<input type="checkbox"/> No
Are the necessary parameters for comparison discussed?	
Is the study referring to semantic clones explicitly?	
Sampling and subject system	
What was the subject system?	<input type="checkbox"/> Yes
– Was the subject system size considerable?	<input type="checkbox"/> No
– Has the researcher mentioned how the subject system was identified and selected?	

Appendix B. (continued)

Data item	Description
carried out?	software clone detection, i.e. recall, precision, application area, scalability, portability, etc.
Subject system	How the data was collected: it refers to the subject system and its size
Data analysis	Data analysis, i.e. corresponding source representation and match detection technique are extracted
Developer of the tool and usage	It refers to the clone detection tool, developer and usage of the tool
Study findings	Major findings or conclusions from the primary study like percentage of cloned code
Other	Does the study explicitly refer to semantic clone detection or model based clone detection, any other important point

A.4. Detailed questions

Section – 4

Findings	
Did the study mention the type of clone?	<input type="checkbox"/> Yes
Consider:	<input type="checkbox"/> No
How well the clones are categorized?	
Did the study explicitly mentions the type of clone, or is to be inferred from the study?	
Level of usage	
Was the tool reported?	<input type="checkbox"/> Yes
Consider:	<input type="checkbox"/> No
Was the study referring to development of a new tool or usage of the tool for analysis of a different subject system?	

Appendix B. Data items extracted from all papers

Data item	Description
Study identifier	Unique ID for the study
Date of data extraction	
Bibliographic data	Author, year, title, source
Type of article	Journal article, conference article, workshop paper
Study aims/context/application domain	What are the aims of the study, i.e. search focus, i.e. the research areas the paper focus on
Study design	Classification of study – clone analysis, clone visualization, survey, comparative analysis, etc.
What is the clone detection technique	It explicitly refers to the clone detection technique and type of clone
How was comparison	Values of important parameters for

Appendix C. Journals/conferences reporting most clone related research

Publication channel	J/C/W	#	N
International Conference on Software Maintenance	C	20	8
International Conference of Software Engineering	C	18	10
International Workshop on Software Clones	W	18	6
Working Conference on Reverse Engineering	C	15	10
International Conference on Program Comprehension	C	12	5
International Workshop on Source Code Analysis and Manipulation	W	11	2
European Conference on Software Maintenance and Reengineering	C	9	4
Object-Oriented Programming, Systems, Languages & Applications	C	8	2
Asia Pacific Software Engineering Conference	C	7	4
Journal of Systems and Software	J	7	2
IEEE Transactions on Software Engineering	J	6	3
Journal of Software Maintenance and Evolution: Research and Practice	J	6	3
European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering	C	5	3
Conference of the Center for Advanced Studies on Collaborative Research	C	5	3
International Conference on Automated Software Engineering	C	5	1
Empirical Software Engineering Journal	J	4	1
International Symposium on Software Metrics	C	4	1
IEEE International Workshop on Website Evolution	W	3	2
Mining Software Repositories	C	3	1
International Workshop On Partial Evaluation And Program Manipulation	W	2	2

(continued on next page)

Appendix C. (continued)

Publication channel	J/C/ W	#	N
Electronic Notes in Theoretical Computer Science	J	2	1
International Symposium on Empirical Software Engineering	C	2	1
International Workshop on Defects in Large Software Systems	W	2	1

J – Journal, C – Conference, W – Workshop, N – Number of studies reporting software clone detection as prime study, # – Total number of articles investigated.

Appendix D. Acronyms

ADT	Abstract Data Type
AOP	Aspect Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
CeDAR	Clone Detection, Analysis and Refactoring
ConQAT	Continuous Quality Assessment Toolkit
CPU	Central Processing Unit
CRD	Centre for Reviews and Dissemination
DP	Dynamic Programming
FIM	Frequent Itemset Mining
GPU	Graphics Processing Unit
HTML	Hyper Text Markup Language
ICA	Independent Component Analysis
IDE	Integrated Development Environment
IEEE	The Institute of Electrical and Electronics Engineers
LCS	Longest Common Subsequence
LOC	Lines of Code
LSI	Latent Semantic Indexing
LSH	Locality Sensitive Hashing
PDG	Program Dependence Graph
RTF	Repeated Tokens Finder
STL	Standard Template Library
UML	Unified Modeling Language
XML	Extensible Markup Language
XVCL	XML Variant Configuration Language
XMI	XML Metadata Interchange

References

- [1] E. Adar, M. Kim, SoftGUESS: visualization and exploration of code clones in context, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 762–766.
- [2] R. Al-Ekram, C. Kapsner, R. Holt, M. Godfrey, Cloning by accident: An empirical study of source code cloning across software systems, in: Proceedings of International Symposium on Empirical Software Engineering (ISESE'05), Noosa Heads, Australia, 2005, pp. 376–385.
- [3] G. Antoniol, G. Cassaza, M. Di Penta, E. Merlo, Modeling clones evolution through time series, in: Proceedings of the 17th International Conference on Software Maintenance (ICSM '01), 2001, pp. 273–280.
- [4] G. Antoniol, U. Villano, E. Merlo, M. Di Penta, Analyzing cloning evolution in the Linux kernel, Information and Software Technology 44 (13) (2002) 755–765.
- [5] L. Aversano, G. Canfora, A. De Lucia, P. Gallucci, Web site reuse: cloning and adapting, in: Proceedings of the 3rd International Workshop on Web Site Evolution (WSE'01), Florence, Italy, 2001, pp. 107–111.
- [6] L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: an empirical study, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, 2007, pp. 81–90.
- [7] B. Baker, On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95), Toronto, Ontario, Canada, 1995, pp. 86–95.
- [8] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, K. Kontogiannis, Measuring clone based reengineering opportunities, in: Proceedings of the 6th International Software Metrics Symposium (METRICS'99), Boca Raton, Florida, USA, 1999, pp. 292–303.
- [9] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, K. Kontogiannis, Partial redesign of Java software systems based on clone analysis, in: Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99), Atlanta, GA, USA, 1999, pp. 326–336.
- [10] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, K. Kontogiannis, Advanced clone-analysis to support object-oriented system refactoring, in: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, 2000, pp. 98–107.
- [11] M. Balint, T. Gîrba, R. Marinescu, How developers copy, in: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06), Athens, Greece, 2006, pp. 56–68.
- [12] T. Bakota, R. Ferenc, T. Gyimóthy, Clone smells in software evolution, in: Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07), Paris, France, 2007, pp. 24–33.
- [13] L. Barbour, H. Yuan, Y. Zou, A technique for just-in time clone detection, in: Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10), Washington DC, USA, 2010, pp. 76–79.
- [14] H. Basit, S. Jarzabek, Detecting higher-level similarity patterns in programs, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'05), Lisbon, Portugal, 2005, pp. 156–165.
- [15] H. Basit, D. Rajapakse, S. Jarzabek, Beyond templates: a study of clones in the STL and some general implications, in: Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA, 2005, pp. 15–21.
- [16] H. Basit, S. Puglisi, W. Smyth, A. Turpin, S. Jarzabek, Efficient token based clone detection with flexible tokenization, in: Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, 2007, pp. 513–515.
- [17] H. Basit, S. Jarzabek, A data mining approach for detecting higher-level clones in software, IEEE Transactions on Software Engineering 35 (4) (2009) 497–514.
- [18] H. Basit, U. Ali, S. Jarzabek, Viewing simple clones from a structural clones' perspective, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 1–8.
- [19] Project Bauhaus, <<http://www.bauhaus-stuttgart.de>> (accessed April 2012).
- [20] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (9) (2007) 577–591.
- [21] N. Bettenburg, W. Shang, W.M. Ibrahim, B. Adams, Y. Zou, A.E. Hassan, An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming 74 (7) (2010) 1–17.
- [22] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenance (ICSM '98), Bethesda, Maryland, USA, 1998, pp. 368–378.
- [23] B. Biegel, S. Diehl, JCCD: a flexible and extensible API for implementing custom code clone detectors, in: Proceedings of 25th International Conference on Automated Software Engineering, (ASE'10), Antwerp, Belgium, 2010, pp. 167–168.
- [24] B. Biegel, S. Diehl, Highly configurable and extensible code clone detection, in: Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, 2010, pp. 237–241.
- [25] P. Brereton, B.A. Kitchenham, D. Budgen, M. Turner, M. Khalil, Lessons from applying the systematic literature review process within the software engineering domain, The Journal of Systems and Software 80 (4) (2007) 571–583.
- [26] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, R. Robbes, Language independent clone detection applied to plagiarism detection, in: Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10), Timisoara, Romania, 2010, pp. 77–86.
- [27] C. Brown, S. Thompson, Clone detection and elimination for Haskell, in: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10), Madrid, Spain, 2010, pp. 111–120.
- [28] M. Bruntink, Aspect mining using clone class metrics, in: Proceedings of the 1st Workshop on Aspect Reverse-Engineering Held in Conjunction with 11th Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 2004, p. 5.
- [29] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourwe, On the use of clone detection for identifying crosscutting concern code, IEEE Transactions on Software Engineering 31 (10) (2005) 804–818.
- [30] D. Budgen, P. Brereton, Performing systematic literature reviews in software engineering, in: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, 2006, pp. 1051–1052.
- [31] P. Bulychiev, M. Minea, Duplicate code detection using anti-unification, in: Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering, St. Petersburg, Russia, 2008, pp. 51–54.

- [32] E. Burd, J. Bailey, Evaluating clone detection tools for use during preventative maintenance, in: *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, Montreal, Canada, 2002, pp. 36–43.
- [33] J. Carver, D. Chatterji, N. A. Craft, On the need for human-based empirical validation of techniques and tools for code clone analysis, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 61–62.
- [34] G. Cassaza, G. Antoniol, U. Villano, E. Merlo, M. Di Penta, Identifying clones in the Linux kernel, in: *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01)*, Florence, Italy, 2001, pp. 90–97.
- [35] M. Chilowicz, É. Duris, G. Roussel, Finding similarities in source code through factorization, *Electronic Notes in Theoretical Computer Science* 238 (5) (2009) 47–62.
- [36] M. Chilowicz, É. Duris, G. Roussel, Syntax tree fingerprinting for source code similarity detection, in: *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, 2009, pp. 243–247.
- [37] S. Choi, H. Park, H. Lim, T. Han, A static API birthmark for windows binary executables, *The Journal of Systems and software* 82 (5) (2009) 862–873.
- [38] E. Choi, N. Yoshida, T. Ishio, K. Inoue, T. Sano, Extracting code clones for refactoring using combinations of clone metrics, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 7–13.
- [39] P. Ciancarini, G.P. Favini, Detecting clones in game playing software, *Entertainment Computing* 1 (2009) 9–15.
- [40] A. Corazza, S. D. Martino, V. Maggio, G. Scanniello, A tree kernel based approach for clone detection, in: *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, 2010, pp. 1–5.
- [41] J.R. Cordy, T.R. Dean, N. Synnysky, Practical language-independent detection of near-miss clones, in: *Proceedings of the 14th IBM Centre for Advanced Studies Conference (CASCON'04)*, Toronto, Ontario, Canada, 2004, pp. 1–12.
- [42] M. Dagenais, E. Merlo, B. Laguë, D. Proulx, Clones occurrence in large object oriented software packages, in: *Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON'98)*, Toronto, Ontario, Canada, 1998, pp. 192–200.
- [43] A.M. Dalgarno, Jump starting software product lines with clone detection, in: *Proceedings of 12th International Software Product Line Conference*, Limerick, Ireland, 2008, pp. 351.
- [44] Y. Dang, S. Ge, R. Huang, D. Zhang, Code clone detection experience at Microsoft, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 63–64.
- [45] I.J. Davis, M.W. Godfrey, From whence it came: detecting source code clones by analyzing assembler, in: *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, Beverly, MA, USA, 2010, pp. 242–246.
- [46] M. D. Wit, A. Zaidman, A. van Deursen, Managing code clones using dynamic change tracking and resolution, in: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, AB, 2009, pp. 169–178.
- [47] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J. Girard, S. Teuchert, Clone detection in automotive model-based development, in: *Proceedings of 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 603–612.
- [48] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, P. Mas, M. Pizka, Tool support for continuous quality control, *IEEE Software* 25 (5) (2008) 60–67.
- [49] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfahler, B. Schaeetz, Model clone detection in practice, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 37–44.
- [50] M. Di Penta, Evolution doctor: a framework to control software system evolution, in: *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, 2005, pp. 280–283.
- [51] M. Di Penta, M. Neteler, G. Antoniol, E. Merlo, A language independent software renovation framework, *The Journal of Systems and Software* 77 (3) (2005) 225–240.
- [52] C. Domann, E. Juergens, J. Streit, The curse of copy & paste-cloning in requirements specifications, in: *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, Lake Buena Vista, Florida, USA, 2009, pp. 443–446.
- [53] E. Duala-Ekoko, M. Robillard, Tracking code clones in evolving software, in: *Proceedings of 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, 2007, pp. 158–167.
- [54] E. Duala-Ekoko, M. Robillard, CloneTracker: tool support for code clone management, in: *Proceedings of 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 843–846.
- [55] E. Duala-Ekoko, M. Robillard, Clone region descriptors: representing and tracking duplication in source code, *ACM Transactions on Software Engineering and Methodology* 20 (1) (2010) 1–31.
- [56] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, Oxford, England, UK, 1999, pp. 109–119.
- [57] S. Ducasse, O. Nierstrasz, M. Rieger, On the effectiveness of clone detection by string matching, *Journal on Software Maintenance and Evolution: Research and Practice* 18 (1) (2006) 37–58.
- [58] W.S. Evans, C.W. Fraser, F. Ma, Clone detection via structural abstraction, *Software Quality Journal* 17 (4) (2009) 309–330.
- [59] R. Falke, P. Frenzel, R. Koschke, Empirical evaluation of clone detection using syntax suffix trees, *Empirical Software Engineering* 13 (6) (2008) 601–643.
- [60] R. Fanta, V. Rajlich, Removing clones from the code, *Journal of Software Maintenance, Research and Practice* 11 (4) (1999) 223–243.
- [61] M. Fowler, K. Beck, T. Brant, W. Opdyke, D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman, 1999.
- [62] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, H. Iida, Code clone graph metrics for detecting diffused code clones, in: *Proceedings of the 16th Asia Pacific Software Engineering Conference (APSEC'09)*, Penang, Malaysia, 2009, pp. 373–380.
- [63] M. Gabel, L. Jiang, Z. Su, Scalable detection of semantic clones, in: *Proceedings of 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 321–330.
- [64] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, Z. Su, Scalable and systematic detection of buggy inconsistencies in source code, in: *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, Nevada, USA, 2010, pp. 175–190.
- [65] K. Gallagher, L. Layman, Are decomposition slices clones? In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, Portland, Oregon, USA, 2003, pp. 251–256.
- [66] D.M. German, M. Di Penta, Y.-G. Gueheneuc, G. Antoniol, Code siblings: technical and legal implications of copying code between applications, in: *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR'09)*, Vancouver, BC, Canada, 2009, pp. 81–90.
- [67] D. Gitchell, N. Tran, Sim: a utility for detecting similarity in computer programs, *ACM SIGCSE Bulletin* 31 (1) (1999) 266–270.
- [68] N. Göde, R. Koschke, Incremental clone detection, in: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany, 2009, pp. 219–228.
- [69] N. Göde, Evolution of Type-1 clones, in: *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, Edmonton, Canada, 2009, pp. 77–86.
- [70] N. Göde, Clone Removal: Fact or Fiction, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 22–40.
- [71] N. Göde, J. Harder, Clone stability, in: *Proceedings of 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, 2011, pp. 65–74.
- [72] N. Göde, J. Harder, Oops!... I changed it again, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 14–20.
- [73] M. W. Godfrey, D. M. German, J. Davies, A. Hindle, Determining the provenance of software artifacts, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 65–66.
- [74] S. Grant, J. R. Cordy, Vector space analysis of software clones, in: *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09)*, Vancouver, BC, Canada, 2009, pp. 233–237.
- [75] J. Guo Y. Zou, Detecting clones in business applications, in: *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, Antwerp, Belgium, 2008, pp. 91–100.
- [76] Y. Hayase, Y. L. Lee, K. Inoue, A criterion for filtering code clone related bugs, in: *Proceedings of the workshop on Defects in large software systems in companion to International Symposium on Software Testing and Analysis (DEFECTS '08)*, Seattle, Washington, 2008, pp. 37–38.
- [77] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, On software maintenance process improvement based on code clone analysis, in: *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES '02)*, Rovaniemi, Finland, 2002, pp. 185–197.
- [78] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, ARIES: refactoring support environment based on code clone analysis, in: *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, MA, USA, 2004, pp. 222–229.
- [79] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, ARIES: Refactoring support tool for code clone, in: *Proceedings of 3rd Workshop on Software Quality in companion to International Conference on Software Engineering (ICSE'05)*, St. Louis, Missouri, USA, 2005, pp. 1–4.
- [80] Y. Higo, Y. Ueda, S. Kusumoto, K. Inoue, Simultaneous modification support based on code clone analysis, in: *Proceedings of the 14th Asia Pacific Software Engineering Conference (APSEC'07)*, Nagoya, Japan, 2007, pp. 262–269.
- [81] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, Method and implementation for investigating code clones in a software system, *Information and Software Technology* 49 (9–10) (2007) 985–998.
- [82] Y. Higo, S. Kusumoto, K. Inoue, A metric based approach to identifying refactoring opportunities for merging code clones in a Java software system, *Journal of Software Maintenance and Evolution: Research and Practice* 20 (6) (2008) 435–461.
- [83] Y. Higo, K. Sawa, S. Kusumoto, Problematic code clones identification using multiple detection results, in: *Proceedings of the 16th Asia Pacific Software Engineering Conference (APSEC'09)*, Penang, Malaysia, 2009, pp. 365–372.
- [84] Y. Higo, S. Kusumoto, Code clone detection on specialized PDG's with heuristics, in: *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, Oldenburg, Germany, 2011, pp. 75–84.
- [85] S. Horwitz, Identifying the semantic and textual differences between two versions of a program, in: *Proceedings of the ACM SIGPLAN'90 Conference on*

- Programming Language Design and Implementation (PLDI'90), White Plains, New York, 1990, pp. 234–245.
- [86] D. Hou, F. Jacob, P. Jablonski, Exploring the design space of proactive tool support for copy-and-paste programming, in: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09), Ontario, Canada, 2009, pp. 188–202.
- [87] D. Hou, P. Jablonski, F. Jacob, CnP: Towards an environment for the proactive management of copy-and-paste programming, in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, BC, Canada, 2009, pp. 238–242.
- [88] D. Hou, F. Jacob, P. Jablonski, Proactively managing copy-and-paste induced code clones, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, AB, 2009, pp. 391–392.
- [89] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: Incremental, distributed, scalable, in: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, 2010, pp. 1–9.
- [90] B. Hummel, E. Juergens, D. Steidl, Index-based model clone detection, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 21–27.
- [91] T. Imai, Y. Kataoka, T. Fukaya, Evaluating software maintenance cost using functional redundancy metrics, in: Proceedings of 26th Annual International Computer Software and Applications (COMPSAC'02), Oxford, England, 2002, pp. 299–306.
- [92] P. Jablonski, Managing the copy-and-paste programming practice in modern IDE's, in: Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Quebec, Canada, 2007, pp. 933–934.
- [93] P. Jablonski, D. Hou, CREn: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE, in: Proceedings of Eclipse Technology Exchange Workshop at OOPSLA 2007 (ETX'07), Montreal, Quebec, Canada, 2007, p. 5.
- [94] P. Jablonski, D. Hou, Aiding software maintenance with copy and paste clone awareness, in: Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10), Washington DC, USA, 2010, pp. 170–179.
- [95] F. Jacob, D. Hou, P. Jablonski, Actively comparing clones inside the code editor, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 1–8.
- [96] K. Jalbert, J. S. Bradbury, Using clone detection to identify bugs in concurrent software, in: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, 2010, pp. 1–5.
- [97] S. Jarzabek, S. Li, Unifying clones with a generative programming technique: a case study, *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons 18 (4) (2006) 267–292.
- [98] Z. M. Jiang, A. E. Hassan, R. C. Holt, Visualizing clone cohesion and coupling, in: Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06), Bangalore, India, 2006, pp. 467–476.
- [99] L. Jiang, Z. Su, E. Chiu, Context-based detection of clone-related bugs, in: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, 2007, pp. 55–64.
- [100] L. Jiang, G. Mishergii, Z. Su, S. Glondou, DECKARD: Scalable and accurate tree-based detection of code clones, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 96–105.
- [101] Z.M. Jiang, A.E. Hassan, A framework for studying clones in large software systems, in: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07), Paris, France, 2007, pp. 203–212.
- [102] L. Jiang, Z. Su, Automatic mining of functionally equivalent code fragments via random testing, in: Proceedings of 18th International Symposium on Software Testing and Analysis (ISSTA'09), Chicago, Illinois, USA, 2009, pp. 81–92.
- [103] Jian-lin Huang, Fei-peng Li, Quick similarity measurement of source code based on suffix array, in: Proceedings of International Conference on Computational Intelligence and Security, Beijing, China, 2009, pp. 308–311.
- [104] J.H. Johnson, Substring matching for clone detection and change tracking, in: Proceedings of the 10th International Conference on Software Maintenance, Victoria, British Columbia, Canada, 1994, pp. 120–126.
- [105] E. Juergens, F. Deissenboeck, B. Hummel, CloneDetective – a workbench for clone detection research, in: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 603–606.
- [106] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? In: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 485–495.
- [107] E. Juergens, N. Göde, Achieving accurate clone detection results, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 1–8.
- [108] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, J. Streit, Can clone detection support quality assessments of requirement specifications? in: Proceedings of 32nd International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 2010, pp. 79–88.
- [109] E. Juergens, F. Deissenboeck, B. Hummel, Code similarities beyond copy & paste, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10), Madrid, Spain, 2010, pp. 78–87.
- [110] E. Juergens, Research in cloning beyond code: a first roadmap, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 67–68.
- [111] W. Jung, C. Wu, E. Lee, WSIM: detecting clone pages based on 3-levels of similarity clues, in: Proceedings of 9th IEEE/ACIS International Conference on Computer and Information Sciences, Yamagata, Japan, 2010, pp. 702–707.
- [112] T. Kamiya, F. Ohata, K. Kundou, S. Kusumoto, K. Inoue, Maintenance support tools for JAVA Programs: CCFinder and JAAT, in: Proceedings of 23rd International Conference on Software Engineering (ICSE'01), Toronto, Ontario, Canada, 2001, pp. 837–838.
- [113] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multi-linguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering* 28 (7) (2002) 654–670.
- [114] T. Kamiya, The Official CCFinderX website <<http://www.ccfinder.net>> (accessed April 2012).
- [115] T. Kamiya, C. Ghezzi, How code skips over revisions, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 69–70.
- [116] C.J. Kapsner, M.W. Godfrey, Aiding comprehension of cloning through categorization, in: Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPS'04), Kyoto, Japan, 2004, pp. 85–94.
- [117] C.J. Kapsner, M.W. Godfrey, Improved tool support for the investigation of duplication in software, in: Proceedings of the 21st International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 2005, pp. 305–314.
- [118] C.J. Kapsner, M.W. Godfrey, Supporting the analysis of clones in software systems: a case study, *Journal of Software Maintenance and Evolution: Research and Practice* 18 (2) (2006) 61–82.
- [119] C.J. Kapsner, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, P. Weisgerber, Subjectivity in clone judgment: can we ever agree? in: Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings, 2007.
- [120] C.J. Kapsner, M.W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, *Empirical Software Engineering* 13 (6) (2008) 645–692.
- [121] S. Kawaguchi, P.K. Garg, M. Matsushita, K. Inoue, Automatic categorization algorithm for evolvable software archive, in: Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPS'03), Helsinki, Finland, 2003, pp. 195–200.
- [122] M. Kim, L. Bergman, T. Lau, D. Notkin, An Ethnographic study of copy and paste programming practices in OOPL, in: Proceedings of 3rd International ACM-IEEE Symposium on Empirical Software Engineering (ISESE'04), Redondo Beach, CA, USA, 2004, pp. 83–92.
- [123] M. Kim, D. Notkin, Using a clone genealogy extractor for understanding and supporting evolution of code clones, in: Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05), Saint Louis, Missouri, USA, 2005, pp. 1–5.
- [124] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, An Empirical study of code clone genealogies, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE 2005'05), Lisbon, Portugal, 2005, pp. 187–196.
- [125] H. Kim, Y. Jung, S. Kim, and K. Yi, MeCC: Memory comparison-based clone detector, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Honolulu, Hawaii, 2011, pp. 301–310.
- [126] M. Kim, Understanding and aiding code evolution by inferring change patterns, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 101–102.
- [127] B. A. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University, Keele and Department of Computer Science, University of Durham, Durham, UK, 2007, p. 65.
- [128] B. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering – a systematic literature review, *Information and Software Technology* 51 (1) (2009) 7–15.
- [129] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, B.V. Saranya, Detection of Type-1 and Type-2 code clones using textual analysis and metrics, in: Proceedings of 2010 International Conference on Recent Trends in Information, Telecommunication and Computing, Kochi, Kerala, India, 2010, pp. 241–243.
- [130] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis (SAS'01), vol. LNCS 2126, Paris, France, 2001, pp. 40–56.
- [131] K. Kontogiannis, Partial design recovery using dynamic programming, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'94), Toronto, Ontario, Canada, 2004, p. 34.
- [132] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, M. Bernstein, Pattern matching for clone and concept detection, *Automated Software Engineering* 3 (1–2) (1996) 77–108.
- [133] K. Kontogiannis, Evaluation experiments on the detection of programming patterns using software metrics, in: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'97), Amsterdam, The Netherlands, 1997, pp. 44–54.

- [134] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, 2006, pp. 253–262.
- [135] R. Koschke, Survey of research on software clones, in: *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007, p. 24.
- [136] R. Koschke, Frontiers of software clone management, in: *Proceedings of Frontiers of Software Maintenance (FoSM'08)*, Beijing, China, 2008, pp. 119–128.
- [137] D. Kozlov, J. Koskinen, M. Sakkinen, J. Markkula, Exploratory analysis of the relations between code cloning and open source software quality, in: *Proceedings of 7th International Conference on the Quality of Information and Communications Technology*, Porto, Portugal, 2010, pp. 358–363.
- [138] J. Krinke, Identifying similar code with program dependence graphs, in: *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, Stuttgart, Germany, 2001, pp. 301–309.
- [139] J. Krinke, A study of consistent and inconsistent changes to code clones, in: *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*, Vancouver, BC, Canada, 2007, pp. 170–178.
- [140] J. Krinke, N. Gold, Y. Jia, D. Binkley, Distinguishing copies from originals in software clones, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 41–48.
- [141] J. Krinke, N. Gold, Y. Jia, D. Binkley, Cloning and Copying between GNOME projects, in: *Proceedings of 7th IEEE International Conference on Mining Software Repositories*, Cape Town, SA, 2010, pp. 98–101.
- [142] J. Krinke, Is cloned code more stable than non-cloned code, in: *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, Beijing, China, 2008, pp. 57–66.
- [143] J. Krinke, Is cloned code older than non-cloned code? in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 28–33.
- [144] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, in: *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, 1997, pp. 314–321.
- [145] A. Lakhota, J. Li, A. Walenstein, Y. Yang, Towards a clone detection benchmark suite and results archive, in: *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, Portland, Oregon, USA, 2003, pp. 285–286.
- [146] F. Lanubile, T. Mallardo, Finding function clones in web applications, in: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, Benevento, Italy, 2003, pp. 379–386.
- [147] T. Lavoie, M. Eilers-Smith, E. Merlo, Challenging cloning related problems with GPU-based algorithms, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 25–32.
- [148] T. Lavoie, E. Merlo, Automated type-3 clone oracle using Levenshtein metric, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 34–40.
- [149] S. Lee, I. Jeong, SDD: High performance code clone detection system for large scale source code, in: *Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA Companion '05)*, San Diego, CA, USA, 2005, pp. 140–141.
- [150] H. Lee, K. Doh, Tree-pattern-based duplicate code detection, in: *Proceedings of International Workshop on Data-intensive Software Management and Mining*, Philadelphia, PA, USA, 2009, pp. 7–12.
- [151] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, S. Kim, Instant code clone search, in: *Proceedings of 18th International Symposium on Foundations of Software Engineering*, NY, USA, 2010, pp. 167–176.
- [152] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: finding copy-paste and related bugs in large-scale software code, *IEEE Transactions on Software Engineering* 32 (3) (2006) 176–192.
- [153] H. Li, S. Thompson, Clone detection and removal for Erlang/OPT within a refactoring environment, in: *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, Savannah, GA, USA, 2009, pp. 169–178.
- [154] Z. O. Li, J. Sun, A metric space based software clone detection approach, in: *Proceedings of 2nd International Conference on Software Engineering and Data Mining*, Chengdu, China, 2010, pp. 111–116.
- [155] H. Liu, Z. Ma, L. Zhang, W. Shao, Detecting duplications in sequence diagrams based on suffix trees, in: *Proceedings 13th Asia-Pacific Software Engineering Conference (APSEC'06)*, Bangalore, India, 2006, pp. 269–276.
- [156] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, Analysis of the Linux kernel evolution using code clone coverage, in: *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN, USA, 2007, pp. 22.
- [157] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder, in: *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, 2007, pp. 106–115.
- [158] A. Lozano, M. Wermelinger, B. Nuseibeh, Evaluating the harmfulness of cloning: A change based experiment, in: *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN, USA, 2007, p. 4.
- [159] A. Lozano, M. Wermelinger, Assessing the effects of clones on changeability, in: *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing, China, 2008, pp. 227–236.
- [160] A. Lozano, M. Wermelinger, Tracking clones' imprint, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 65–72.
- [161] G.A. Lucca, M. Di Penta, A.R. Fasolino, An approach to identify duplicated web pages, in: *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02)*, Oxford, England, 2002, pp. 481–486.
- [162] A. Lucia, R. Francese, G. Scanniello, G. Tortora, Reengineering web applications based on cloned pattern analysis, in: *Proceedings of 12th International Workshop on Program Comprehension (IWPC'04)*, Bari, Italy, 2004, pp. 132–141.
- [163] A. Lucia, R. Francese, G. Scanniello, G. Tortora, Understanding cloned patterns in web applications, in: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, St. Louis, MO, USA, 2005, pp. 333–336.
- [164] A. Lucia, M. Risi, G. Tortora, G. Scanniello, Clustering Algorithms and latent semantic indexing to identify similar pages in web applications, in: *Proceedings of the 9th International Workshop on Web Site Evolution (WSE'07)*, Paris, France, 2007, pp. 65–72.
- [165] Y. Ma, D. Woo, Applying a code clone detection method to domain analysis of device drivers, in: *Proceedings of the 14th Asia Pacific Software Engineering Conference (APSEC'07)*, Nagoya, Japan, 2007, pp. 254–261.
- [166] K. Maeda, Syntax sensitive and language independent detection of code clones, *World Academy of Science, Engineering and Technology* 60 (2009) 350–354.
- [167] A. Y. Mao, J. R. Cordy, T. R. Dean, Automated conversion of table-based websites to structured stylesheets using table recognition and clone detection, in: *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'07)*, Richmond Hill, Ontario, Canada, 2007, pp. 12–26.
- [168] A. Marcus, J. I. Maletic, Identification of high-level concept clones in source code, in: *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, CA, USA, 2001, pp. 107–114.
- [169] D. Martin, J. R. Cordy, Analyzing web service similarities using contextual clones, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 41–46.
- [170] J. Mayrand, C. Leblanc, E.M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, Monterey, CA, USA, 1996, pp. 244–253.
- [171] T. Mende, F. Beckwermer, R. Koschke, G. Meier, Supporting the Grow-and-Prune model in software product lines evolution using clone detection, in: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, Szeged, Hungary, 2008, pp. 163–172.
- [172] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, Software quality analysis by code clones in industrial legacy software, in: *Proceedings of 8th IEEE International Symposium on Software Metrics (METRICS'02)*, Ottawa, Canada, 2002, pp. 87–94.
- [173] A. Monden, S. Okahara, Y. Manabe, K. Matsumoto, Guilty or not guilty: using clone metrics to determine open source licensing violations, *IEEE Software* 28 (2) (2011) 42–47.
- [174] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen, Cleman: comprehensive clone group evolution management, in: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, 2008, pp. 451–454.
- [175] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen, Clone-aware configuration management, in: *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009, pp. 123–134.
- [176] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen, ClemanX: Incremental clone detection tool for evolving software, in: *Proceedings of 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, 2009, pp. 437–438.
- [177] T.T. Nguyen, H.A. Nguyen, J.M. Al-Kofahi, N.H. Pham, T.N. Nguyen, Scalable and incremental clone detection for evolving software, in: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, Edmonton, AB, 2009, pp. 491–494.
- [178] J.R. Pate, R. Tairas, N.A. Craft, Clone Evolution: A Systematic Review, Technical Report SERG-2010-01, University of Alabama, Alabama, USA, 2010, p. 20.
- [179] J.-F. Patenaude, E. Merlo, M. Dagenais, B. Laguë, Extending software quality assessment techniques to Java systems, in: *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, Pittsburgh, PA, 1999, pp. 49–56.
- [180] A. Perumal, S. Kanmani, E. Kodhai, Extracting the similarity in detected software clones using metrics, in: *Proceedings of International Conference on Computer and Communication Technology*, 2010, Allahabad, Uttar Pradesh, India, pp. 575–579.
- [181] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, in: *Proceedings of 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08)*, 2008, Bari, Italy, pp. 71–80.
- [182] N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Complete and accurate clone detection in graph based models, in: *Proceedings of 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, 2009, pp. 276–286.

- [183] W. Qu, Y. Jia, M. Jiang, Pattern mining of cloned codes in software systems, *Information Sciences* 180 (2010) 1–11.
- [184] F. Rahman, C. Bird, P. Devanbu, Clones: what is that smell, in: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South, Africa, 2010, pp. 72–81.
- [185] D. Rajapakse, S. Jarzabek, An investigation of cloning in web applications, in: *Proceedings of the Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW'05)*, Chiba, Japan, 2005, pp. 924–925.
- [186] D. Rajapakse, S. Jarzabek, Using server pages to unify clones in web applications: A trade-off analysis, in: *Proceedings of the 29th International Conference of Software Engineering (ICSE'07)*, Minneapolis, USA, 2007, pp. 116–126.
- [187] C.K. Roy, J.R. Cordy, A Survey on Software Clone Detection Research, Technical Report 2007-541, Queen's University at Kingston Ontario, Canada, 2007, p. 115.
- [188] C.K. Roy, J.R. Cordy, NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 172–181.
- [189] C.K. Roy, J.R. Cordy, Scenario-based comparison of clone detection techniques, in: *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 153–162.
- [190] C.K. Roy, J.R. Cordy, A mutation/injection-based automatic framework for evaluating code clone detection tools, in: *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshops*, Denver, Colorado, USA, 2009, pp. 157–166.
- [191] C.K. Roy, Detection and analysis of near miss software clones, in: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, AB, 2009, pp. 447–450.
- [192] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: a qualitative approach, *Science of Computer Programming* 74 (7) (2009) 470–495.
- [193] C.K. Roy, J.R. Cordy, Near miss function clones in open source software: an empirical study, *Journal of Software Maintenance and Evolution: Research and Practice* 22 (3) (2010) 165–189.
- [194] C.K. Roy, J.R. Cordy, Are scripting languages really different, in: *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, 2010, pp. 17–24.
- [195] F. V. Rysseberghe, S. Demeyer, Evaluating clone detection techniques from a refactoring perspective, in: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, Linz, Austria, 2004, pp. 336–339.
- [196] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, Z. Su, Detecting code clones in binary executable, in: *Proceedings of International Symposium on Software Testing and Analysis*, Chicago, Illinois, USA, 2009, pp. 117–127.
- [197] R.K. Saha, M. Asaduzzaman, M.F. Zibran, C.K. Roy, K.A. Schneider, Evaluating code clone genealogies at release level: An empirical study, in: *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, Timisoara, Romania, 2010, pp. 87–96.
- [198] R.K. Saha, C.K. Roy, K.A. Schneider, Visualizing the evolution of code clones, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 71–72.
- [199] Y. Sasaki, T. Yamamoto, Y. Hayase, K. Inoue, Finding file clones in FreeBSD ports collection, in: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South, Africa, 2010, pp. 102–105.
- [200] S. Schulze, M. Kuhlemann, M. Rosenmüller, Towards a refactoring guideline using code clone classification, in: *Proceedings of 2nd Workshop on Refactoring Tools with companion to Conference on Object Oriented Programming Systems Languages and Applications*, Nashville, Tennessee, 2008, p. 4.
- [201] S. Schulze, S. Apel, C. Kastner, Code clones in feature oriented software product lines, in: *Proceedings of Generative Programming and Component Engineering (GPCE'10)*, Eindhoven, The Netherlands, 2010, pp. 103–112.
- [202] P. Schugert, Scalable clone detection using description logic, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 47–53.
- [203] Gehan M.K. Selim, K.C. Foo, Y. Zou, Enhancing source based clone detection using intermediate representation, in: *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, Beverly, MA, USA, 2010, pp. 227–236.
- [204] Gehan M.K. Selim, L. Barbour, W. Shang, B. Adams, A.E. Hassan, Y. Zou, Studying the impact of clones on software defects, in: *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, Beverly, MA, USA, 2010, pp. 13–21.
- [205] D.M. Shawky, A.F. Ali, An approach for assessing similarity metrics used in metric based clone detection techniques, in: *Proceedings on 3rd IEEE International Conference on Computer Science and Information Technology*, Chengdu, China, 2010, pp. 580–584.
- [206] D.M. Shawky, A.F. Ali, Modeling clones evolution in open source systems through chaos theory, in: *Proceedings on 2nd International Conference on Software Technology and Engineering*, San Juan, Puerto Rico, 2010, pp. VI-159–VI-164.
- [207] Tool Simian <<http://www.harukizaemon.com/simian/index.html>> (accessed April 2012).
- [208] Tool SimScan <<http://www.blue-edge.bg/download.html>> (accessed April 2012).
- [209] H. Storrle, Towards clone detection in UML domain models, in: *Proceedings of European Conference on Software Architecture (ECSA'10)*, Copenhagen, Denmark, 2010, pp. 285–293.
- [210] A. Sutton, H. Kagdi, J.I. Maletic, G. Volkert, Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones, in: *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'05)*, Washington DC, USA, 2005, pp. 1079–1080.
- [211] N. Synytskyy, J.R. Cordy, T. Dean, Resolution of static clones in dynamic web pages, in: *Proceedings of 5th IEEE International Workshop on Web Site Evolution (WSE'03)*, Amsterdam, The Netherlands, 2003, pp. 49–56.
- [212] R. Tairas, Clone detection and refactoring, in: *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, 2006, pp. 780–781.
- [213] R. Tairas, J. Gray, I. D. Baxter, Visualization of clone detection results, in: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, Portland, Oregon, USA, 2006, pp. 50–54.
- [214] R. Tairas, J. Gray, Phoenix-based clone detection using suffix trees, in: *Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE'06)*, Melbourne, Florida, USA, 2006, pp. 679–684.
- [215] R. Tairas, J. Gray, I.D. Baxter, Visualizing clone detection results, in: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, Atlanta, Georgia, USA, 2007, pp. 549–550.
- [216] R. Tairas, Clone maintenance through analysis and refactoring, in: *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*, Atlanta, GA, USA, 2008, pp. 29–32.
- [217] R. Tairas, Centralizing clone group representation and maintenance, in: *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications Companion to the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, Orlando, Florida, USA, 2009, pp. 781–782.
- [218] R. Tairas, J. Gray, Get to know your clones with CeDAR, in: *Proceedings of Conference on Object Oriented Programming Systems Languages and Applications Companion to 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009, Orlando, Florida, USA, pp. 817–818.
- [219] R. Tairas, J. Gray, An information retrieval process to aid in the analysis of code clones, *Empirical Software Engineering* 14 (1) (2009) 33–56.
- [220] R. Tairas, J. Gray, Sub-clone refactoring in open source software artifacts, in: *Proceedings of ACM Symposium on Applied Computing (SAC'10)*, Sierre, Switzerland, 2010, pp. 2373–2374.
- [221] R. Tairas, F. Jacob, J. Gray, Representing clones in a localized manner, in: *Proceedings of 5th International Workshop on Software Clones*, Honolulu, USA, 2011, pp. 54–60.
- [222] R. Tiarks, R. Koschke, R. Falke, An assessment of type-3 clones as detected by state-of-the-art tools, in: *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, Edmonton, Canada, 2009, pp. 67–76.
- [223] S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta, An empirical study on the maintenance of source code clones, *Empirical Software Engineering* 15 (1) (2010) 1–34.
- [224] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, On detection of gapped code clones using gap locations, in: *Proceedings 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, Gold Coast, Queensland, Australia, 2002, pp. 327–336.
- [225] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, Gemini: Maintenance support environment based on code clone analysis, in: *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, Ottawa, Canada, 2002, pp. 67–76.
- [226] V. Wahler, D. Seipel, J.W. Gudenberg, G. Fischer, Clone detection in source code by frequent itemset techniques, in: *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04)*, Chicago, IL, USA, 2004, pp. 128–135.
- [227] A. Walenstein, N. Jyoti, J. Li, Y. Yang, A Lakhotia, Problems creating task-relevant clone detection reference data, in: *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, Victoria, BC, Canada, 2003, pp. 285–295.
- [228] R. Wettel, R. Marinescu, Archeology of code duplication: Recovering duplication chains from small duplication fragments, in: *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2005, p. 8.
- [229] J. Whaley, M.S. Lam, Cloning- based context- sensitive pointer alias analysis using binary decision diagrams, in: *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'04)*, Washington DC, USA, 2004, pp. 131–144.
- [230] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, H. Iida, SHINOBI: a tool for automatic code clone detection in the IDE, in: *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, Lille, France, 2009, pp. 313–314.
- [231] W. Yang, Identifying syntactic differences between two programs, *Software Practice and Experience* 21 (7) (1991) 739–755.
- [232] R. Yokomori, H. Siy, N. Yoshida, M. Noro, K. Inoue, Measuring the effects of aspect-oriented refactoring on component relationships: two case studies, in:

- Proceedings of 10th Aspect-Oriented Software Development Conference (AOSD'11), Pernambuco, Brazil, 2011, pp. 215–226.
- [233] N. Yoshida, T. Ishio, M. Matsushita, K. Inoue, Retrieving similar code fragments based on identifier similarity for defect detection, in: Proceedings of the 2008 workshop on Defects in large software systems in companion to International Symposium on Software Testing and Analysis (DEFECTS'08), Seattle, Washington, 2008, pp. 41–42.
- [234] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, On refactoring support based on code clone dependency relation, in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), Como, Italy, 2005, pp. 16–25.
- [235] N. Yoshida, T. Hattori, K. Inoue, Finding similar defects using synonymous identifier retrieval, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 49–56.
- [236] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, M. Low, Query-based filtering and graphical view generation for clone analysis, in: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08), Beijing, China, 2008, pp. 376–385.