JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.579

Clone evolution: a systematic review

Jeremy R. Pate[1], Robert Tairas[2] and Nicholas A. Kraft[1,*,†]

[1]*Department of Computer Science, The University of Alabama, Tuscaloosa, AL, USA*
[2]*AtlanMod Team (INRIA & EMN), École des Mines de Nantes, Nantes, France*

SUMMARY

Detection of code clones — similar or identical source code fragments — is of concern both to researchers and to practitioners. An analysis of the clone detection results for a single source code version provides a developer with information about a discrete state in the evolution of the software system. However, tracing clones across multiple source code versions permits a clone analysis to consider a temporal dimension. Such an analysis of clone evolution can be used to uncover the patterns and characteristics exhibited by clones as they evolve within a system. Developers can use the results of this analysis to understand the clones more completely, which may help them to manage the clones more effectively. Thus, studies of clone evolution serve a key role in understanding and addressing issues of cloning in software. In this paper, we present a systematic review of the literature on clone evolution. In particular, we present a detailed analysis of 30 relevant papers that we identified in accordance with our review protocol. The review results were organized to address three research questions. Through our answers to these questions, we present the methods that researchers have used to study clone evolution, the patterns that researchers have found evolving clones to exhibit, and the evidence that researchers have established regarding the extent of inconsistent change undergone by clones during software evolution. Overall, the review results indicate that whereas researchers have conducted several empirical studies of clone evolution, there are contradictions among the reported findings, particularly regarding the lifetimes of clone lineages and the consistency with which clones are changed during software evolution. We identify human-based empirical studies and classification of clone evolution patterns as two areas that are in particular need of further work. Copyright © 2011 John Wiley & Sons, Ltd.

Received 20 December 2010; Revised 29 August 2011; Accepted 8 September 2011

KEY WORDS: code clones; clone evolution; systematic literature review

1. INTRODUCTION

Code clones are source code fragments that are similar or identical in terms of text, vocabulary, structure, or meaning. Fowler *et al*. [1] classified code duplication (cloning) as a bad smell and thus as a significant indicator of poor software maintainability. However, more recent work indicates that clones are not as harmful as previously believed [2,3] and actually may improve productivity [4]. Indeed, Rahman *et al*. [5] find little empirical evidence that clones negatively affect software maintainability but do find that cloned code may be less fault prone than that of non-cloned code. Still, there are long-term risks associated with cloning, such as the potential duplication of defects and the possible loss of (implicit or explicit) links among code fragments that must remain consistent [6]. Thus, the research community continues to study code clones and their implications.

Two surveys published in 2007 provide overviews of the code-clone literature. Koschke [7] summarizes important results from the field and presents open questions, whereas Roy and Cordy [8] thoroughly review the field, albeit with a strong emphasis on clone detection. Although the

*Correspondence to: Nicholas Kraft, Department of Computer Science, The University of Alabama, Tuscaloosa, AL, USA.
†E-mail: nkraft@cs.ua.edu

evolution of code clones throughout the history of a software system must be understood to appreciate the full impact of those clones on the system, the topic of clone evolution is discussed only briefly in the aforementioned surveys. Consequently, Harder and Göde [9] provide a concise survey of existing methods to model clone evolution and present open questions not yet addressed.

Analysis of clone evolution can reveal patterns and characteristics exhibited by clones as they evolve within a system. For example, such an analysis can reveal which clones result from copy-paste activity, which clones are change-prone, and which clones are long-lived. Developers can use this information to manage clones more effectively. For example, because developers tend to lose dependent associations between copied and pasted code [10], special documentation effort or tool support may be needed to prevent the loss of these associations. Further, stable and long-lived clones are likely well tested and may be poor candidates for refactoring [11]. Short-lived ("young") clones may similarly be poor candidates for refactoring, whereas refactoring change-prone clones may reduce software maintenance cost and effort.

In this paper, we present a systematic review of 30 primary studies, which focus on clone evolution. The overall goal of our systematic literature review [12,13] is to identify the role of code clones during software evolution. Thus, to address that goal, we developed three research questions. Before presenting these research questions in Section 1.2, we first provide background information about code clones, including their detection and evolution.

1.1. Background

A *code fragment* is any sequence of source code lines (or comment lines) at any level of granularity. Researchers studying code clones typically consider the sequence of statements or the function/method body level of granularity. A code fragment can be identified uniquely by its enclosing file name, beginning line number, and ending line number. A more robust alternative for uniquely identifying a code fragment is the *clone region descriptor* (CRD) [14], which identifies a code fragment using software metrics and syntactic, structural, and lexical information.

A code fragment is a *clone* of another code fragment if they are similar with respect to syntax or semantics, although there is no consensus regarding what constitutes "similar" [7,15]. Conceptually, given a similarity function s and a similarity threshold T, code fragments F_1 and F_2 are similar if $s(F_1, F_2) \geq T$. Any code fragment that is similar to another is considered a clone, two clones form a *clone pair*, and three or more clones form a *clone class* or *clone group*. Each clone in a clone group must be mutually similar to every other clone in the group.

For more complete definitions of terms and techniques related to code clones and their detection, refer to surveys by Koschke [7] and by Roy and Cordy [8]. For comprehensive evaluations of code-clone detection techniques and tools, refer to studies by Bellon *et al*. [16] and by Roy *et al*. [17]. In particular, Bellon *et al*. [16] use eight open source software systems to provide a quantitative comparison and evaluation of six clone detectors, and Roy *et al*. [17] use a unified conceptual framework to provide a qualitative comparison and evaluation of over 40 clone-detection techniques and tools. Finally, Tairas provides a comprehensive code clones bibliography online.[1]

Clone evolution is the study of clones as they evolve across versions of the source code. Researchers studying clone evolution generally observe how clones change from one version to the next. For example, a *clone genealogy* traces the lineage and change history of a clone group across versions of the source code. For a particular clone group and version, a clone genealogy is a directed acyclic graph that links the clone group to all corresponding clone groups in a subsequent version, and so on across the versions of interest [2]. Constructing a clone genealogy is difficult, because each individual clone in one version must be mapped to the corresponding clone in a subsequent version, and the clones in a clone group can undergo inconsistent (independent) changes over time [3,18]. A *fragment trace* is similar to a clone genealogy but is defined at the level of individual clones rather than that of clone groups [6]. Note that "version" might indicate a *revision* created by committing the source code for a software system to a repository (e.g., using CVS or Subversion) or a *release* designated by the developers of a software system.

[1] http://www.cis.uab.edu/tairasr/clones/literature

DOI: 10.1002/smr

1.2. Research questions

We address three research questions in this paper. The first research question (*RQ1*) is motivated by the desire to understand how researchers study clone evolution. That is, we want to identify and understand the methods that researchers have used for the purpose of studying clone evolution. The second research question (*RQ2*) is motivated by the desire to understand the results of the studies that researchers have conducted. That is, we want to identify and understand the patterns of clone evolution that researchers have found using the methods identified in the answer to *RQ1*. The third research question (*RQ3*) is motivated by one of the main problems assumed to be caused by clones. In particular, many researchers have motivated the need for clone detection and clone comprehension by noting that if the clones in a clone group are changed inconsistently, a defect could be introduced into the system. Thus, we want to understand whether and how frequently inconsistent changes occur, and clone evolution studies can help us to do so.

The three research questions that we address in this paper are:

RQ1: What methods have been used to study clone evolution?
RQ2: What patterns do evolving clones exhibit?
RQ3: What evidence is there that clones change (in)consistently during software evolution?

The answer to *RQ1* will describe the methods that researchers have used to study clone evolution. For empirical studies, we document the sources of the collected data. That is, we document whether developer behavior was observed or source code was analyzed. Further, we identify the techniques/tools used to detect clones and the software system(s) used as subject(s). For other studies, we document the characteristics of the proposed approach. For example, we document whether an approach is meant to be applied to an existing software system (i.e., whether the approach mines the system history from a repository) or whether it is meant to be applied to a new system (i.e., whether the approach performs analysis only for periods during which it is in use).

The answer to *RQ2* will describe the change patterns of clone evolution that researchers have identified using empirical studies. The discussion of RQ2 (in Section 3.2) is organized according to the kind of empirical study used to discover clone evolution patterns. In particular, two kinds of studies emerged from our review: those focused on developer behavior and those focused on clone lineage. In addition to documenting the clone evolution change patterns, we document the parameters of the studies used to identify those patterns. That is, we document the techniques used to map clones between source code revisions, the techniques used to evaluate those mappings, and the amounts of time used to space source code revisions.

The answer to *RQ3* will describe the evidence that researchers have established about whether and to what extent clones undergo consistent change during software evolution. Researchers track changes to clones either at the fragment level or at the group level, but in any case, consistent and inconsistent changes are measured at the clone group level. As defined by Kim *et al*. (SLR 15), a *consistent change* to a clone group is one in which the clone fragments in the clone group are updated uniformly. Similarly, an *inconsistent change* to a clone group is one in which one or more clone fragments in the clone group are updated divergently. After reporting the evidence, we report correlations between (in)consistent changes to clones and system quality that researchers have discovered or conjectured.

1.3. Organization

The remainder of this paper is organized as follows. In Section 2, we describe our systematic literature review protocol, and in Section 3, we report the results of the systematic literature review. In Section 4, we discuss the overall findings, and in Section 5, we conclude.

2. REVIEW PROTOCOL

A systematic literature review [12,13] is a formal, repeatable method by which to identify, evaluate, and interpret the available research related to a research question or topic area. Conducting a

DOI: 10.1002/smr

systematic review involves gathering primary studies from which to synthesize conclusions. Common uses of such reviews include verification/refutation of prior claims, identification/investigation of research gaps, and motivation/contextualization of new research. The key benefit of conducting a systematic review, rather than an ad hoc review, is in the added confidence provided to the authors and readers regarding the completeness of the gathered evidence.

A systematic review is performed as follows:

- Plan the review
  - Identify the need for a systematic review
  - Develop a focused research question
  - Formulate a protocol around that research question
- Conduct the review
  - Search the databases for primary studies
  - Evaluate the primary studies for relevance and quality
  - Extract data from the primary studies
- Report the results
  - Synthesize and summarize the extracted data
  - Interpret the results with respect to the research question
  - Write the report

The protocol for our systematic review comprises research goals and questions, source selection and search, primary study identification and evaluation, and data extraction and synthesis. In Section 1.2, we described the research goals and questions, and in Sections 2.1–2.3, we provide an overview of the remainder of the protocol. To reduce researcher bias, one author developed the protocol, and another author reviewed it. Before approving the final version, we discussed, reviewed, and evolved the protocol.

2.1. Source selection and search

We obtained the primary studies used in this systematic review by searching databases that contain research in software maintenance and evolution. In particular, we selected databases that: (i) contain peer-reviewed journal articles, conference proceedings, and book chapters; (ii) overlap in content as little as possible; and (iii) appear in other systematic reviews on software engineering topics. The selected databases are: ACM Digital Library, Google Scholar, IEEE Xplore Digital Library, ScienceDirect (Elsevier), SpringerLink, and Wiley InterScience.

From our research questions, we constructed a global search string using the following strategy:

- Extract major terms from the research questions
- Generate a list of synonyms and alternate spellings for the major terms
- Combine each major term with its synonyms and alternate spellings using Boolean *OR*
- Link major terms using Boolean *AND*

The resulting global search string was: ((‘code’ *OR* ‘software’ *OR* ‘application’) *AND* (‘clone’ *OR* ‘cloning’ *OR* ‘copy’ *OR* ‘duplicate’ *OR* ‘duplication’ *OR* ‘similarity’) *AND* (‘change’ *OR* ‘evolution’ *OR* ‘genealogy’ *OR* ‘maintenance’ *OR* ‘management’ *OR* ‘tracking’)) For each of the selected databases, we formed a query from the global search string based on the input structure required by the database interface.

2.2. Primary study identification and evaluation

From the results of our database searches, we identified the primary studies for this systematic review. We traversed each result list, evaluating each paper in turn. We continued to traverse the list until encountering five consecutive papers of no relevance to the research questions.

DOI: 10.1002/smr

We defined the inclusion and exclusion criteria listed in Table I and applied those criteria to each paper to evaluate its relevance. Application of the inclusion/exclusion criteria is performed as follows:

1. Use the title to exclude any paper that is clearly not related to the research questions
2. Use the abstract and keywords to exclude any paper that has a relevant title but is not related to the research questions
3. Read the paper and include it if any of the research questions are addressed

Table II lists the distribution of the primary studies, including the number of primary studies from each source.

2.3. Data extraction and synthesis

To extract relevant data from each primary study, we designed and used a data extraction form. The form includes a number of fields, although not every paper provided data for every field. The fields from the data extraction form and their descriptions are listed in Table III. The first author reviewed each primary study paper, recording relevant data to a data extraction form, and another author independently reviewed and extracted data from a small sample set of primary studies. This procedure is similar to those used in other systematic reviews [19–22]. We then compared the extracted data for consistency, which would indicate agreement in interpretation. Indeed, the consistency among the extracted data suggested that the data extraction forms completed by the first author were sufficient. Finally, we synthesized the data extracted from the primary studies to answer the research questions posed in this systematic review.

3. RESULTS

In this section, we analyze the 30 primary studies that we identified in accordance with our review protocol. Our analysis is partitioned into three subsections, each of which addresses one research question. Appendix C provides a complete listing of the primary studies, which are labeled SLR 1 through SLR 30.

3.1. RQ1: What methods have been used to study clone evolution?

The majority of the 30 papers (97%) include either an empirical study,[2] or a review of prior empirical studies. Specifically, 27 papers (90%) include at least one study in which source code was analyzed, two papers (7%) (SLR 6, 14) include a study in which developer behavior was observed, and one paper (3%) (SLR 12) does not include an empirical study. For the 27 papers including a study in which source code was analyzed, we list in Appendix A and Appendix B the clone detector(s) used and the software system(s) studied, respectively. In total, we list 14 clone detectors and 58 subject systems in these appendices.

Three of the primary studies are unique in that they focus primarily on developer behavior. Two of these papers (SLR 6, 14) describe findings obtained via the direct or virtual observation of developers as they created/edited source code. In particular, Kim *et al.* (SLR 14) conducted two studies on the copy-paste programming practices of developers using object-oriented programming languages. In the first study, Kim *et al.* directly observed four developers for about 10 h of programming in Java/C++/Jython, manually logging edit operations (copy, cut, paste, delete, undo, and redo) and asking the developers to state their intentions for copy-and-pasting as these operations were performed. In the second study, Kim *et al.* virtually observed five developers for 50 h of programming in Java. Using an Eclipse plug-in, Kim *et al.* tracked edits made to the source code by developers and later replayed the edits using the plug-in, first inferring the programmers' intentions for copying-and-pasting, and next interviewing each programmer twice to confirm the accuracy of their inferences. Based on their observations, they concluded that developers tended to wait until after several copy-and-paste

[2]In an empirical study, observation is used to collect data, and that data is analyzed quantitatively or qualitatively.

DOI: 10.1002/smr

Table I. Inclusion/exclusion criteria.

Inclusion criteria	Exclusion criteria
• Journal articles, conference proceedings, white and book chapters	• Preliminary conference versions of included white journal articles
• Primary sources of survey papers	• Papers not in English
• Empirical studies (quantitative or qualitative) white on code-clone evolution	• Survey papers, introductions to special issues, white short papers, and tutorials
• Papers that describe techniques and tools white for managing code clones in evolving software	• Papers focused on code-clone detection

Table II. Distribution of the 30 primary studies.

Source	Acronym	Count
IEEE International Conference on Software Maintenance	ICSM	4
IEEE International Working Conference on Source Code Analysis and Manipulation	SCAM	3
IEEE Working Conference on Mining Software Repositories	MSR	3
IEEE International Conference on Program Comprehension	ICPC	2
International Workshop on Software Clones	IWSC	2
Working Conference on Reverse Engineering	WCRE	2
ACM Transactions on Software Engineering and Methodology	TOSEM	1
Empirical Software Engineering: An International Journal	ESE	1
IEEE Transactions on Software Engineering	TSE	1
Information and Software Technology	IST	1
Journal of Software Maintenance and Evolution: Research and Practice	JSME	1
Systems and Computers in Japan	SCJ	1
ACM SIGSOFT International Symposium on the Foundations of Software Engineering	FSE	1
Asia-Pacific Software Engineering Conference	APSEC	1
European Conference on Software Maintenance and Reengineering	CSMR	1
IEEE International Symposium on Software Metrics	METRICS	1
IEEE/ACM International Conference on Automated Software Engineering	ASE	1
International Conference on Fundamental Approaches to Software Engineering	FASE	1
International Conference on Software Engineering	ICSE	1
International Symposium on Empirical Software Engineering	ISESE	1

Table III. Data extraction form.

Category	Data item	Description
Metadata	Identifier	Unique identifier
	Database	ACM/Google/IEEE/Elsevier/Springer/Wiley
	Bibliographic	Author, title, venue, year
	Topic(s)	Major focus area(s) and general subject area(s)
Clone detector	Type	Text, token, syntax, or semantics based
	Tool	Name, version, configuration
Implementation	Other techniques	Technique(s) used in combination with clone detection
	Environment	Development tools with which the technique is integrated (e.g., Eclipse)
Availability	Tool	Availability of tool implementing technique (i.e., license and URL)
	Data	Availability of test data, including (preprocessed) input and results
Studies	Design	Type of study (e.g., comparative or statistical) and experimental setup
	Purpose	Goals of the study (i.e., research questions addressed)
	Evaluation	How the results were evaluated (i.e., metrics used)
	Findings	Major findings and conclusions

operations before attempting to restructure the code. However, they also noted that, over time, developers tended to lose the dependent associations between copied and pasted text.

In the second paper, focused primarily on developer behavior, de Wit *et al*. (SLR 6) provided developers with CloneBoard, an Eclipse plug-in that provides copy-paste replacement operators

DOI: 10.1002/smr

inspired by those proposed by Mann [23]. de Wit *et al*. observed the behavior of developers using the tool to perform tasks designed with the intent of provoking code cloning. In the third paper, Balint *et al*. (SLR 4) describe a tool, Clone Evolution View that provides visualization of code clone evolution that includes edit information graphed over time, with color codes corresponding to different developers. The result is an author-centric view of clone evolution and the ways in which a common multiple-developer environment manifests itself in clone evolution. Balint *et al*. used their visualization tool to conduct a retrospective study on three Java systems and concluded that developers are relevant variables to consider when analyzing code clones (and their evolution).

Three other primary studies are unique in that they focus primarily on tools to assist developers in tracking or managing evolving clones. Two of these tools, CloneTracker and CnP, are implemented as Eclipse plug-ins, and the third tool, Clever, is implemented as an extension to Subversion (via Subclipse, an Eclipse plug-in providing support for Subversion). Duala-Ekoko and Robillard (SLR 7) developed CloneTracker to demonstrate the efficacy of the CRD for representing a clone independent of its exact text or location in a file. CloneTracker builds custom clone documentation for a system and provides the option for linked editing [24] of all clones in a clone group. Thus, CloneTracker provides tool support for ensuring consistent modification of related clones. Hou *et al*. (SLR 12) designed and implemented CnP to manage copy-paste programming. By tracking clones from their creation, the CnP could be used to control the evolution of clones throughout the lifetime of a software system, although Hou *et al*. do not present an empirical study. Nguyen *et al*. (SLR 24) developed Clever to make source code management clone-aware. Unlike other clone management tools, Clever tracks clone changes at the revision level. Clever manages clone evolution via a number of clone operations, including detection, change management, consistency validation, synchronization, and merging.

Of the remaining 24 primary studies, the majority focus on the study of patterns exhibited by evolving clones. The basic approach to discovering a clone evolution pattern is to collect clone detection results for an initial source code version and then to trace the detected clones (at varying levels of granularity, e.g., clone or clone group) across subsequent versions of the source code. Among all methods that researchers have used to study clone evolution, tracing clones across versions is by far the most common. For example, Göde and Koschke (SLR 11) extend their incremental clone detection algorithm to detect the ancestor of each clone for a given revision. Using these so-called fragment traces, Göde and Koschke explore the volatility of clones by measuring the lifetimes of fragment traces.

3.2. RQ2: What patterns do evolving clones exhibit?

Understanding of the patterns that emerge from analysis of the clones in an evolving software system is required to accurately model the characteristics of evolving clones. Over the lifetime of a system, clones can be created, changed, or eliminated. Changes to the cloned code impact the containing clone groups. The potential impact of changes to cloned code on the correctness and completeness of bug fixes and feature additions highlights the need for code clone awareness. Information about patterns and characteristics of evolving code clones can inform refactoring (and other system reengineering) efforts, for example, by indicating whether a clone group or set of clone groups is likely to be removed during normal evolution activities or should be targeted for refactoring before "aging" occurs, impeding future evolution activities.

Based on our review, we have identified two distinct kinds of studies, which aim to discover clone evolution patterns. In the first kind of study, the focus of the discovered pattern(s) is developer behavior, and in the second kind, the focus is clone lineage, which comprises fragment traces and clone genealogies. The remainder of this section is organized according to pattern focus.

3.2.1. Developer behavior. The observation of developer behavior regarding clones provides information on why developers create clones, how they use them, and how they maintain them. These patterns of cloning are important components to consider when attempting to understand the results observed in other studies, which focus solely on the source code of the subject systems. These observed patterns provide context, and provide another perspective from which to view clone evolution.

Kim *et al*. (SLR 14) conducted an ethnographic study of copy-and-paste programming practices, the design of which we described in Section 3.1. Based on their observations, Kim *et al*. concluded that developers tended to wait until after several copy-and-paste operations before attempting to restructure

DOI: 10.1002/smr

the code. However, they also noted that, over time, developers tended to lose the dependent associations between copied and pasted text. Thus, they recommended the development and use of clone management tools that:

- Visualize copy-paste contents
- Maintain copy-paste dependencies
- Learn structural templates from repeated instances of the replicate and specialize pattern (in which code is cloned and then customized to introduce a new feature)
- Warn developers who attempt to change a structural template
- Suggest refactorings and present structural templates from which to choose

Balint *et al.* (SLR 4) conducted experiments in which they sought to empirically detect what they termed "cloning activity patterns" in three open source Java systems. According to Balint *et al.*, cloning activity patterns result from three developer activities: (i) *line cloning*, which is the introduction of a single new line to a clone or clone group, (ii) *block cloning*, which is the introduction of multiple lines to a clone or clone group (or possibly the introduction of a new clone or clone group), and (iii) *line fixing*, which is the introduction of a single new line to a clone to bring it into consistency with the rest of its clone group. Moreover, each of the developer activities may be characterized by one of two attributes: *consistent*, which denotes a line or block cloning in which all clones in the affected clone group(s) are updated uniformly, and *inconsistent*, which denotes a line or block cloning in which one or more clones in the affected clone group(s) are updated divergently. A line fixing activity is always the result of a prior inconsistent line or block cloning activity.

By analyzing the visualizations generated by their tool (Clone Evolution View), Balint *et al.* identified five cloning activity patterns:

- Consistent line/block cloning with unique author
- Creation of clones by multiple authors using consistent block cloning
- Consistent line/block cloning with multiple authors
- Inconsistent line cloning fixed by same author
- Inconsistent line cloning fixed by different authors

3.2.2. Clone lineage. There are four basic approaches to constructing clone genealogies (or similarly, fragment traces). In the first approach, clones are detected in all source code versions of interest, and then corresponding clones in consecutive versions are retroactively linked. This approach requires the use of a heuristic to determine whether a clone in version *i* is indeed the modified version of a clone in version *i-1* [9]. We review three studies (SLR 3, 15, 26) that used this first approach. In the second approach, clones are detected in an initial source code version of interest, and then the clones are traced across versions using change information mined from a repository or IDE [9]. A disadvantage of this approach is that clones introduced in versions following the initial version are not detected. We review two studies (SLR 2, 27) that used this second approach. The third approach to constructing clone genealogies is to combine the first two approaches [9]. In particular, clones are detected in all source code versions of interest, the clones are transformed to CRDs [14], the CRDs are traced across versions, and the textual differences between the clone regions are detected and stored. We review one study (SLR 5) that used this third approach. Finally, in the fourth approach, clones are mapped during clone detection using information about changes between versions. The mapping is performed at the code fragment level, allowing analysis of code fragment evolution patterns as well as analysis of clone group evolution patterns. We review two studies (SLR 10, 11) that used this fourth approach.

Kim *et al.* (SLR 15) presented an empirical study of the genealogical traits of code clones in two open source Java systems and defined the following evolution patterns for clone groups: same ($CG_{i-1} = CG_i$), add (≥ 1 clone in CG_i is not in CG_{i-1}), subtract (≥ 1 clone in CG_{i-1} is not in CG_i), consistent change (all clones in CG_{i-1} changed consistently), inconsistent change (≥ 1 clone in CG_{i-1} changed inconsistently), and shift (≥ 1 clone in CG_i partially overlaps with ≥ 1 clone in CG_{i-1}). They developed a tool to automate extraction of clone genealogies over the lifetime of a software system. In particular, their clone genealogy extractor, CGE, takes as input: (i) a sequence of software system revisions in chronological order; (ii) a clone detector; and (iii) a code snippet location tracker. To perform their study, Kim *et al.* instantiated CGE with CCFinder (to detect clones) and Kenyon [25] (to trace code

DOI: 10.1002/smr

snippet locations across software system revisions). Kim *et al.* manually evaluate the extracted clone genealogies.

Bakota *et al.* (SLR 3) proposed a technique for determining the mapping of clone groups across revisions while accounting for the evolution of the constituent clones. The application of a similarity measure they developed to compute the mappings informed their definitions of "clone smells." They apply the similarity measure to pairs of clones from consecutive versions if those clones have the same type of "head" AST node. That is, they apply the measure to two clones at the same granularity (e.g., method or class). Given the two clones, the similarity measure considers the clones' lexical similarity (measured using Levenshtein distance), relative positions, AST ancestor node names, and file names. They manually evaluate mappings of interest, that is, those mappings contributing to the clone smells.

In all, Bakota *et al.* (SLR 3) defined four clone smells: vanished clone instance (VCI), occurring clone instance (OCI), moving clone instance (MCI), and migrating clone instance (MGCI). The VCI smell indicates changes to or deletions of a code fragment that make it no longer a part of its original clone group. Conversely, introducing a new code fragment into a clone group causes the OCI smell. The MCI and MGCI smells indicate a code fragment joining a different clone group and moving back to the original clone group, respectively.

Bakota *et al.* (SLR 3) conducted a case study to validate the usefulness of the clone smells that they defined. The subject of the case study was *Mozilla Firefox*, specifically, 12 consecutive HEAD revisions distributed evenly across the year 2006. Using a tool, which they implemented as part of the Columbus framework, Bakota *et al.* identified 13 VCI instances, 10 OCI instances, four MCI instances, and five MGCI instances. Of these true positive instances, four VCI instances indicated the possible or actual introduction of bugs into the system, and eight OCI instances resulted from fixes to bugs recorded in Bugzilla. Moreover, three MCI instances indicated possible bugs, whereas four MGCI instances were connected to actual bugs.

Saha *et al.* (SLR 26) conducted a study of clone genealogies at the release level, stating that they chose the release level to avoid short-term experimentations of developers. They studied 17 open source systems of varying sizes in four different languages, and concluded that clone evolution is not affected significantly by either the size or the development language of the system. Saha *et al.* observed that many genealogies are long-lived, implying that more clone groups are added than removed as a system matures. Further, they reported that volatile clones, those that are both created and removed during the observation period, typically disappear within the span of a few releases.

To study how clones are maintained, Aversano *et al.* (SLR 2) adopted two of the six code clone evolution patterns defined by Kim *et al.* (SLR 15) (subtract and consistent change) and adapted a third (inconsistent change). They partitioned inconsistent change into two constituent evolution patterns: independent evolution, where two or more clones belonging to the same group evolve differently across revisions, and late propagation, where a change is propagated consistently but asynchronously across clones in the same group. Using these evolution patterns, Aversano *et al.* performed an empirical study on the effect of maintenance activities on the clones in two open source Java systems. They gathered evolution data for each system using SimScan, a syntax-based clone detector, and CVS. In particular, they traced code clones and logical changes — changes that simultaneously impact source code entities in different files — across more than 5 years of the system's lifetime. Like Kim *et al.*, Aversano *et al.* manually evaluate the extracted clone genealogies.

Thummalapenta *et al.* (SLR 27) reported the results of an empirical study in which they classified clones into evolution patterns and investigated the relationships between the presence of those clone evolution patterns and other characteristics of the clones, including size, location, and kinds of changes undergone (e.g., corrective maintenance). They used both token-based and syntax-based clone detectors, along with a composite differencing algorithm, to study two open source Java systems and two open source ANSI C systems. In particular, to perform token-based clone detection, they applied CCFinder to all four systems, and to perform syntax-based clone detection, they applied SimScan to the two Java systems, and Bauhaus ccdiml to the two ANSI C systems. Moreover, to track the evolution of the detected clones (i.e., to identify clone genealogies) they used a differencing approach [26] composed of three differencing algorithms: CVS/Subversion diff, cosine similarity, and Levenshtein distance [27].

DOI: 10.1002/smr

Thummalapenta *et al*. (SLR 27) considered four clone evolution patterns. The following list provides the name of each pattern, along with a description of how the clones within a clone group are changed for that pattern.

- *Consistent evolution (CO)*
  Consistently
- *Late propagation (LP)*
  Inconsistently (realigned within time interval T, where $T > 24$ h)
- *Delayed propagation (L2)*
  Inconsistently (realigned within 24 h)
- *Independent evolution (IE)*
  Inconsistently (evolved divergently throughout time interval T, where $T > 24$ h)

Thummalapenta *et al*. noted that while LP and IE are mutually exclusive within time interval T, a clone group could undergo LP but be misclassified as undergoing IE if the clones are realigned beyond time interval T. However, they also noted that the instances of LP that they observed always occurred in much less time than that of their interval of observation (T), leading them to conclude that such misclassifications would occur only rarely.

Bettenburg *et al*. (SLR 5) reported the results of an empirical study on the effect of inconsistent changes to clone genealogies on software quality at the release level (i.e., the effect on software quality as perceived by the end user). They used SimScan to detect clones, CRDs to abstract information about the detected clones and to help trace them across releases (i.e., discover clone genealogies), and manual inspection to identify inconsistent changes to a clone genealogy and to classify each genealogy into one of eight cloning patterns identified by Kapser and Godfrey [4]. Using these tools and processes, Bettenburg *et al*. studied *Apache Mina* and *jEdit*. In particular, for *Apache Mina*, they analyzed 306 clone genealogies comprising 1387 clone groups in 22 releases, and for *jEdit*, they analyzed 818 clone genealogies comprising 11,160 clone groups in 50 releases.

For *Apache Mina*, Bettenburg *et al*. (SLR 5) reported that 244 of 306 clone genealogies spanned multiple releases, and that the average lifetime for one of those genealogies was 4.59 releases. Further, the average size of a clone genealogy in *Mina* was 2.56 clones, with the largest genealogy containing 14 clones. For *jEdit*, they reported that 746 of 818 clone genealogies spanned multiple releases, and that the average lifetime for one of those genealogies was 9.00 releases. In addition, the average size of a clone genealogy in *jEdit* was 2.13 clones, with the largest genealogy containing 166 clones. Overall, for the two projects, the average lifetime for a clone genealogy was 6.79 releases, and the average size was 2.79 clones.

Göde (SLR 10) conducted a study of nine open source systems to classify evolution patterns of Type 1 clones. Similar to Krinke (SLR 16), Göde conducted his study on 200 revisions, divided by 1-week intervals. To do so, he introduced a new, incremental approach to constructing fragment traces (i.e., clone lineages at the individual clone level of granularity, as opposed to clone genealogies, which are clone lineages at the group level of granularity). The systems studied by Göde comprised three Java systems, three C systems, and three C++ systems. He found clone evolution to be a system specific classifier, that is, the computed metrics varied based on the characteristics of the software system. However, he found general patterns in the data, including that the ratio of clones in the system decreased on average, with the average clone lifetime being greater than 1 year.

Lozano and Wermelinger (SLR 22) conducted a study of "clone imprints" — progressive or lasting effects of clones on a system. They studied five open source Java systems of varying application domains using their clone tracking tool, which is based on CCFinder and is set to detect clones of at least 30 tokens (about three LOCs). They used their tool to find that "[cloned] methods tend to be cloned most of their lifetime" and that "cloned methods have longer lifetimes than methods not cloned." The former finding differs from a finding reported by Kim *et al*. (SLR 15) for *dnsjava*, which states that most clones last less than eight commits. However, Lozano and Wermelinger (SLR 22) state that they have previously identified *dnsjava* as having an unusual clone evolution (SLR 21), which may account for the apparently contradictory results.

Whereas Krinke (SLR 17) reported that lines of code cloned are more stable than lines of code not cloned, Lozano and Wermelinger (SLR 22) found that, among methods which change, cloned methods

DOI: 10.1002/smr

change more often than do methods not cloned. Based on the different results and on the observation that cloned code often accounts for less than 25% of total code, Lozano and Wermelinger conclude that whereas cloned methods change less than do methods not cloned, cloned methods are more likely to change than are methods not cloned. Although they note that Krinke (SLR 17) analyzes only clones of at least 11 LOCs, Lozano and Wermelinger (SLR 22) state that cloned methods are less stable than methods not cloned, and that their results contradict those of Krinke (SLR 17).

Göde and Koschke (SLR 11) again used an incremental approach to detect clones and construct fragment traces. They analyzed 66 revisions of the *Bauhaus program analysis suite*, spaced weekly throughout the period of 1 January 2008 to 1 April 2009. By measuring the lifetimes of the constructed fragment traces, Göde and Koschke addressed the question of whether clones are volatile. In particular, they traced the lifetimes of 2365 clones, computing a mean age of 50.5 revisions (i.e., 50.5 weeks) and a median age of 66 revisions. Based on their data, they conclude that if a clone does not disappear soon after its creation, it is likely to remain in the system for a long time.

Göde and Koschke (SLR 11) compared, at a high level, their findings on clone lifetimes with the findings of Kim *et al*. (SLR 15) on clone genealogy lifetimes. They concluded that their own results are not consistent with those of Kim *et al*. In particular, Göde and Koschke found that the majority of clones in *Bauhaus* survive for at least 66 weeks, whereas Kim *et al*. reported that ≈76% of clone genealogies in *CAROL* survived for less than 50 days, and that ≈72% of clone genealogies in *dnsjava* survived for less than 11 days. Göde and Koschke reported a belief that these discrepancies are caused, at least in part, by *Bauhaus* being much larger than either *CAROL* or *dnsjava*.

3.2.3. Summary. In this section, we summarize the clone evolution patterns from the literature. We first consider patterns that model individual clones across versions. Such patterns can model the *appearance*, *disappearance*, or *reappearance* of a clone between two versions. These patterns can also model the *changes* undergone by a clone between two versions. We observe that changes to a specific code fragment are not the only factor in its appearance, disappearance, or reappearance as a clone. In particular, the addition of (or a change to) a new code fragment could cause another code fragment to appear as a clone, the removal of (or a change to) a code fragment could cause another code fragment to disappear as a clone, or similarly, a removal/change followed by an addition/change operation could cause another code fragment to reappear as a clone.

We next consider patterns that model clone groups across versions. Kim *et al*. (SLR 15) attempted to identify all possible changes to a clone group between two versions, whereas Bakota *et al*. (SLR 3) focused on changes that could affect software quality. Nevertheless, there is significant overlap among the patterns that they identified. Patterns common to both papers are: *add* (SLR 15) or OCI (SLR 3), *subtract* (SLR 15) or VCI (SLR 3), and *shift* (SLR 15) or MCI (SLR 3). Kim *et al*. identified three additional patterns that model clone group changes between two versions: *no change* or same (SLR 15), *consistent change* (SLR 15), and *inconsistent change* (SLR 15). Thummalapenta *et al*. (SLR 27) later substituted the term consistent evolution for consistent change.

We finally consider patterns that model clone groups across more than two versions. Aversano *et al*. (SLR 2) partitioned Kim *et al*.'s (SLR 15) inconsistent change into *independent evolution*, which corresponds to VCI, and late propagation, which corresponds to Bakota *et al*.'s (SLR 3) MGCI. Thummalapenta *et al*. (SLR 27) extended the work of Aversano *et al*. and partitioned late propagation into *delayed propagation*, where clones are realigned within 24 h, and *late propagation*, where clones are realigned within time interval T, where $T > 24$ h. Of course, the distinction between an independent evolution and a delayed/late propagation depends on the period of observation, and an independent evolution could actually be a delayed/late propagation where that period extended.

3.3. RQ3: What evidence is there that clones change (in)consistently during software evolution?

Many of the patterns described in Section 3.2 are defined in terms of the changes undergone by evolving clones. In particular, many patterns are detected by determining whether and to what extent clones change consistently or inconsistently during their lifetimes. Changes are tracked at varying levels of granularity. For example, several clone smells (SLR 3) track changes to clones between two source code versions, whereas the *consistent change* and *inconsistent change* patterns trace

DOI: 10.1002/smr

clone groups between two versions. As defined by Kim *et al*. (SLR 15), *consistent* denotes a line or block cloning in which all clones in the affected clone group(s) are updated uniformly, and *inconsistent* denotes a line or block cloning in which one or more clones in the affected clone group (s) are updated divergently. Further, two patterns defined by Aversano *et al*. (SLR 2), *late propagation* and *independent evolution*, trace changes to clone groups across multiple versions, as do two of the clone smells (MCI and MCGI) defined by Bakota *et al*. (SLR 3). In the remainder of this section, we review the evidence that researchers have established regarding the extent of inconsistent change undergone by evolving clones.

Balint *et al*. (SLR 4) conducted a retrospective study of developer behavior, looking for instances of the cloning activity patterns they defined. Their results showed that the rate of detection of inconsistent changes correlated with the number of developers. Kim *et al*. (SLR 15) focused their study on determining how often programmers update clones consistently, how long clones remain in a system, and what characteristics are exhibited by evolving clones that cannot be removed easily from a system using refactoring techniques. The results of the study led them to reach two conclusions that deviated from conventional wisdom on clones. First, clones with long lifetimes that undergo consistent changes become more difficult to refactor with standard techniques as they "age" in the system, and second, many of the volatile clones in the system were eventually removed through normal evolution and may not have needed or benefited from refactoring. However, as an anonymous reviewer of this paper noted, removal is a refactoring and the volatile clones may have been actively removed. That is, the nature of the volatile clones is unclear, as is the definition of "normal evolution."

Saha *et al*. (SLR 26) collected clone genealogy data for 17 systems and observed that at the release level, 11 to 38% of the genealogies changed consistently over time (about 24% on average). They concluded that this finding is not inconsistent to that of Kim *et al*. (SLR 15), who found that, at the revision level, 36 to 38% of genealogies were changed consistently. Saha *et al*. also observed that about 67% of genealogies among all 17 systems did not undergo line additions, line deletions, or any syntactic changes. Further, they found that such genealogies tended to be long-lived, noting that 69% of them still remained at the end of the observation period.

Of the eight Kapser–Godfrey cloning patterns [4], Bettenburg *et al*. (SLR 5) reported that the majority of long-lived code clones in *Mina* and *jEdit* are instances of the replicate and specialize pattern. This finding is significant, because these clones typically evolve independently, potentially leading to errors caused by reduced awareness of their presence in the system. Nevertheless, Bettenburg *et al*. did not observe a high fraction of errors introduced to replicate-and-specialize clone groups through inconsistent changes. Thus, they concluded that the developers of both *Mina* and *jEdit* are aware of those long-lived clones and are able to manage their independent evolution effectively. Regardless of cloning pattern, Bettenburg *et al*. found a low occurrence of inconsistent changes leading to software errors (ranging from 1.26 to 3.23%), indicating that inconsistent clones are not always indicative of software fault and suggesting that developers are able to manage and control the evolution of clones at the release level.

Aversano *et al*. (SLR 2) found that clones of different granularities exhibited distinct evolution patterns. The majority of class-level clones in their subject systems underwent consistent changes, whereas method-level clones underwent consistent change and independent evolution at nearly identical rates. Aversano *et al*. found that 45 and 74% of clones were changed consistently for *ArgoUML* and *dnsjava*, respectively, although the sample size for *dnsjava* is small. They also found that 19 of the 112 clone groups for *ArgoUML* underwent late propagation, and that in six of those cases, the late propagation was related to bug fixing and thus was potentially harmful. Additionally, they observed that the developers of the systems typically put off disseminating maintenance involving clones unless the maintenance was corrective. Although Aversano *et al*. conclude that "the majority of clone classes is always maintained consistently", their result for *ArgoUML* contradicts this conclusion.

Krinke (SLR 16) performed a large empirical study intended to verify two hypotheses that he formulated based on the results of earlier work by Kim *et al*. (SLR 15) and Aversano *et al*. (SLR 2): (i) code fragments in a clone group are changed consistently during software evolution; and (ii) when code fragments in a clone group are not changed consistently during software evolution,

DOI: 10.1002/smr

the missing changes are made in a later revision of the system. He studied the validity of the hypotheses for five open source systems, using 200 versions (spaced exactly one week apart) for each system. Of the five systems, three were written in Java, one in C, and one in C++.

Krinke (SLR 16) used Simian, a text-based clone detector, to identify clone groups for each system. After running pilot tests, he configured Simian to detect almost identical clones (i.e., Type 2 clones) of at least 11 source code lines. Also, to eliminate spurious changes, Krinke performed two normalization steps on each revision before clone detection: comment removal and pretty printing using Artistic Style.[3] Krinke's result for *ArgoUML* is consistent with that of Aversano *et al*. (SLR 2). Further, Krinke found that clone groups changed consistently roughly half (45–55%) of the time (rejecting the first hypothesis) and that, when clone groups changed inconsistently, they rarely became consistently changed clone groups at a later time (partially rejecting the second hypothesis). In a related study, Krinke (SLR 17) reported that cloned code was more stable that non-cloned code (i.e., there were more additions and deletions to non-cloned code than to cloned code).

Thummalapenta *et al*. (SLR 27) collected several pieces of evidence from their quantitative results. First, they reported that, in cases where consistent evolution was needed, developers often propagated clone changes immediately. Indeed, the results for all four subject systems indicated that less than 16% of clone changes were late propagations. Thummalapenta *et al*. claim that these findings suggest that developers know when and how to propagate clone changes and that refactoring clones to reduce the need for clone change propagation might be unnecessary (as previously suggested by Cordy [28]).

Second, Thummalapenta *et al*. (SLR 27) reported that, other than consistent evolution, independent evolution is the pattern most commonly observed in the four systems. Indeed, in some of the systems, independent evolutions were more common than consistent evolutions. Independent evolutions are often instances of the "replicate and specialize" pattern. Third, Thummalapenta *et al*. reported that neither clone granularity nor the distance between the clones in a clone group appeared to have an effect on the evolution pattern(s) of those clones. The latter observation led them to conclude that developers are effective at tracking clones across multiple areas in a system. Yet, they do not address how and when developers track clones or at what cost. Finally, Thummalapenta *et al*. reported that high proportions of bug fixing changes (i.e., corrective maintenance) affected clone groups that include a lately propagated clone.

Based on the data collected from nine subject systems, Göde (SLR 10) reported that many of the inconsistent clones were not brought into consistency with the clone group via late propagation. However, he observed that consistency of the changes to the clones was primarily system dependent. Thus, he concluded that clone evolution must be investigated for each individual software system according to its own characteristics.

Lozano and Wermelinger (SLR 22) hypothesized that clones have a neglible imprint — that is, that clones are stable with little effect on system degredation over time. They based this hypothesis on previous results, which indicated that clones are stable (SLR 17), and which indicated that clones are volatile (i.e., short-lived) (SLR 15). To test their hypothesis, Lozano and Wermelinger use their tool, which uses a robust tracking approach that handles host method renaming or movement. Further, it automatically detects late propagations, wherein individual clones are changed inconsistently within a clone group but are later returned to consistency via propagation of the change to the inconsistent clone(s). Compared with prior results based on manual late propagation detection (SLR 2), the results based on automatic late propagation detection indicate that clones undergo late propagation less often but undergo independent evolution (wherein individual clones are changed inconsistently within a clone group and evolve independently for the remainder of the study period) more often than previously reported (SLR 2).

Göde and Koschke (SLR 11) observed the changes to individual clones in *Bauhaus*. They use a suffix tree based incremental clone detector to extract Type 1 clones of at least 100 tokens. Göde and Koschke reported that, of the 2365 clones they traced, 82% were unmodified during the 66-week observation period. However, they also reported that, of the 431 that were changed, 81% were changed inconsistently (with the other clones in the clone group). As Göde concluded in earlier work (SLR 10), Göde and Koschke (SLR 11) concluded that the characteristics of clone

[3]http://astyle.sourceforge.net/

DOI: 10.1002/smr

evolution depend on the characteristics of the subject system, making observations about clone evolution difficult to generalize.

Göde and Koschke's results differ from those reported by Krinke (SLR 16) and Aversano *et al.* (SLR 2), which indicate that ≈50% of traced clones were maintained consistently. However, unlike Göde and Koschke, who used suffix tree based incremental clone detection, Krinke and Aversano *et al.* used text-based clone detection (Simian) and syntax-based clone detection (SimScan), respectively. Further, Göde and Koschke studied a commercial system, whereas Krinke and Aversano *et al.* studied open source systems.

Overall, there is evidence of varying degrees of change consistency among the observed clones that changed over time. In particular, we identified the following statistics regarding consistent changes to clones: 11 to 38% of genealogies (SLR 26), 36 to 38% of genealogies (SLR 15), 45 and 74% of clones (SLR 2), and 45 to 55% of clone groups (SLR 16). On the other hand, most studies (SLR 10, 16, 22, 27) conclude that, when clones are changed inconsistently, they are rarely made consistent later. That is, late propagation is rare, which suggests that inconsistently changed clones evolve independently. However, two studies show that the majority of the clones do not change at all: Saha *et al.* (SLR 26) report that 67% of genealogies did not change syntactically and Göde and Koschke (SLR 11) report that 82% of clones were unmodified. In Section 4.2, we discuss the need for comparative studies in clone evolution. Such studies, which would apply different techniques and tools to the same subject systems at the same tracking interval and for the same tracking duration, would help to better understand the influence of the techniques and tools being used to study change consistency in clone evolution.

4. DISCUSSION

Our results indicate that there is great opportunity for further research in clone evolution. In addition to those areas noted by Harder and Göde [9], we believe that there are two primary areas of opportunity for future work in clone evolution: (i) human-based empirical studies; and (ii) classifying patterns. In the former area, we identified only two papers, which indicate a large gap in the literature. In the latter area, we believe that there are two basic questions that must be answered. First, how do internal factors influence the characteristics of clone evolution patterns? Given the various clone detection tools available and the myriad ways in which a clone genealogy or fragment trace can be constructed, we might expect some variation in the clone evolution patterns that are discovered. Studies that control for these internal factors are needed. Second, what system characteristics influence clone evolution patterns? Characteristics that could be studied include software development process used, coding standards enforced, and specific developers involved.

In this section, we highlight future research directions that we identified via our systematic literature review.

4.1. Human-based empirical validation

Most empirical studies reported in the literature are retrospective and are conducted using source code analysis. Such studies are valuable in that they can be used to identify the patterns exhibited by evolving clones. Moreover, these studies can be used to establish evidence about the kinds of changes that evolving clones undergo. However, our results indicate a dearth of human-based empirical studies on clone evolution. Only two such studies (SLR 6, 14) have been reported in the literature, and the first of those studies (SLR 14) predates most of the significant work on clone evolution. Further, to the best of our knowledge, the clone literature contains only one other human-based empirical study. In that study, Chatterji *et al.* [29] provided developers with a CCFinder clone report and observed the developers performing two bug localization tasks, one of which involved locating both instances of the bug — one instance in each of a clone group's two clones.

Conclusions about human behavior must not be drawn only from analytical data. That is, claims about human behavior must be validated using human-based empirical studies. Such studies have been used to understand developer behavior during software engineering tasks such as software

DOI: 10.1002/smr

inspections (e.g., [30]). Indeed, many types of human-based empirical studies have proven useful in different settings and for addressing different types of research questions. These study types range from controlled studies to case studies that allow for naturalistic observation of behavior in practice.

Based on our review of the literature, we have identified the need for human-based empirical studies as a primary area of opportunity for future work in clone evolution [31]. We found little empirical evidence that strongly suggests the usefulness of clone evolution information to developers. Göde and Koschke [11] recently reported that clone lineages and change histories provide important information that permits developers to assess the relevance and threat potential of clones. Specifically, they observed that clones that change infrequently would be worse candidates for refactoring/removal than would be clones that change frequently. However, observation or interviews could be used to better understand whether and how developers would actually use such information. We provide additional examples in the next paragraph.

Several researchers, based on their source code analysis results, have inferred developer knowledge or intent. For example, Bettenburg *et al*. (SLR 5) inferred from their results that the developers of two subject systems were aware of long-lived clones and that those developers were able to effectively manage the independent evolution of those clones. Similarly, Thummalapenta *et al*. (SLR 27) inferred from their results that developers of four subject systems knew when and how to propagate clone changes, and that developers were effective at tracking clones across multiple areas in a system. Two potential human-based empirical studies relate to investigating the inferences of Bettenburg *et al*. (SLR 5) and Thummalapenta *et al*. (SLR 27). Surveys or developer interviews could be used to determine developer awareness of clones or developer knowledge of when and how to propagate clone changes. Observation could be used to determine how developers address clone evolution tasks (e.g., [29]) or how developers track clones across multiple areas of a system. The resulting studies would provide insight to those seeking to understand developer behavior and to those seeking to build or refine tools to assist with that behavior.

Other potential human-based empirical studies relate to investigating fundamental questions in clone evolution, such as the following: When do developers track clones? Do they track clones continuously or as needed (e.g., when late propagating a clone change)? How do developers track clones? What information sources (e.g., repositories or personal experience) do developers use to track clones? What is the cost (in developer time and effort) of tracking clones?

Additional human-based empirical studies relate to investigating the efficacy of clone evolution tools for developers. The goal is to evaluate whether a tool helps developers to be more effective and efficient while performing a change task in the presence of code clones. The baseline against which the new tool is measured could be "no tool", "clone-detection tool", or "competing clone evolution tool". For example, we could use a 2x2 factorial study with repeated measures to compare two of the clone evolution visualizations in Cyclone[4] [32] — for example, Evolution View and Plot View. Moreover, we could use that same study design to compare CnP (SLR 12) to Cyclone. The two tools offer distinct views of clone evolution and distinct interfaces for navigating those views. CnP (SLR 12) actively tracks clones as they are created/modified/removed in the IDE, whereas Cyclone provides an interactive visualization of mined clone lineages. The comparison would also provide insight into the relative advantages and disadvantages of tracking clone evolution interactively (CnP) versus retrospectively (Cyclone).

4.2. Clone evolution patterns: study settings

Our results indicate that most researchers investigated clone evolution patterns by tracing the lineage and change history of clones across source code versions, where each consecutive version is one of: next revision in repository, next revision after set time interval, or next release. Researchers tracing clones across releases argue that doing so reduces the impact of developer experimentation and of other short-term artifacts of the development process. However, when compared with studying clone evolution at the revision level, such an experimental design increases the difficulty of mapping clones between versions and also decreases the precision of the resulting mappings. Further, because

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

DOI: 10.1002/smr

the results of studies tracing clones across releases downplay volatile (i.e., short-lived) clones, they might not be as useful to someone designing a tool for clone tracking or clone management [9].

In Section 3.2.2, we described four basic approaches to constructing clone genealogies or fragment traces. Researchers have used each of the four approaches, and their findings have not always been completely consistent. However, the extent to which these inconsistencies are attributable to the approach used is unknown. Study settings differ substantially, which hinders comparison of results.

Table IV lists, in chronological order, the settings for the clone lineage tracking studies reviewed in Section 3.2.2. The only subject systems that have been studied by multiple research groups are: *ArgoUML*, *CAROL*, *FileZilla*, *jEdit*, and *dnsjava*. It is worth noting that Krinke (SLR 16) rejected the use of *dnsjava* because the changes affecting clones were few and small, and Lozano and Wermelinger (SLR 21) described the clone evolution of *dnsjava* as "unusual". None of the listed systems have been tracked for the same interval and duration by different research groups. For example, three separate groups — Kim *et al*. (SLR 15), Krinke (SLR 16), and Saha *et al*. (SLR 26) — have studied *CAROL*, but those groups studied different numbers of versions (37,[5] 200, and 10, respectively) and tracked clones for different durations (2002–2004, 2002–2006, and 2002–2005, respectively) using different intervals (Revision, Weekly, and Release, respectively).

Table IV makes clear the need for comparative studies in which the following variables are controlled: subject system, tracking interval, and tracking duration. A comprehensive study would mimic the comparative study of clone detectors conducted by Bellon *et al*. [16]. As with the study of clone detectors, the study of clone evolution patterns would compare space and time requirements and would evaluate precision and recall using an oracle. However, whereas the oracle used by Bellon *et al*. contained clones, the clone evolution oracle would contain instances of patterns such as late propagation or independent evolution. Agreement on a set of patterns to include in the oracle should be possible to achieve, as our results indicate that researchers describe clone evolution patterns using a core set of concepts (even though no consensus exists regarding the exhaustiveness of the current list of patterns).

Table IV suggests to us the need for a metric or suite of metrics to measure clone evolution. Given two versions of a system, a metric could consider characteristics of each clone change (such as magnitude or kind — e.g., consistent or inconsistent) or characteristics of each clone group change (such as amount or effect — e.g., late propagation). Such clone evolution metrics could be statistically analyzed to gain new insight into clone evolution patterns. For example, a series of studies using nonparametric simple regression analysis or a single study using a nonparametric multiple regression analysis could be conducted with change as the dependent variable and with possible predictor variables including clone type, clone detector, clone mapper, or tracking interval. A single tool would be applied multiple times to one or more subject systems to gather the data.

4.3. Clone evolution patterns: subject system characteristics

Some researchers have studied multiple software systems using a single approach to constructing genealogies/traces. The lack of discernible system-independent clone evolution patterns in some cases has led these researchers to conclude that clone evolution is a system-specific classifier. However, if clone evolution does follow a system specific pattern, there might be source code metrics that can be used to predict these patterns. Investigating the predictive power of traditional metrics (which are calculated on static program representations such as call graphs or program dependence graphs) seems like an obvious starting point.

Semantic metrics (e.g., [33]) consider the semantic information (i.e., the words) in a system and are often based on topic modeling techniques such as latent semantic indexing [34] or latent Dirichlet allocation [35]. Topic modeling is often applied to source code (e.g., [36]), and has been applied to code clones as well. Tairas and Gray [37] used the semantic information in clones to cluster clone groups detected by CCFinder. Their approach differs from that of Marcus and Maletic [38], who used semantic information in source code to detect (high-level) clones. Tairas and Gray discovered

[5]Note that the 37 versions of *CAROL* studied by Kim *et al*. (SLR 15) were selected from the 164 total versions collected throughout the study duration.

DOI: 10.1002/smr

Table IV. Summary of clone lineage tracking study settings (in chronological order).

Primary study	Subject system	No. of versions	Tracking interval	Tracking duration	
				Start	End
SLR 15	CAROL	37	Revision	Aug 2002	Oct 2004
	dnsjava	224	Revision	Mar 1999	Nov 2004
SLR 2	ArgoUML	5525	Revision	1 Dec 2000	7 Dec 2005
	dnsjava	1200	Revision	27 Mar 1999	26 Jun 2006
SLR 23	dnsjava	N/A	N/A	N/A	N/A
SLR 3	Mozilla Firefox	12	Monthly	Jan 2006	Dec 2006
SLR 16	ArgoUML	200	Weekly	8 Aug 2002	1 Jun 2006
	CAROL	200	Weekly	8 Aug 2002	1 Jun 2006
	Eclipse JDT Core	200	Weekly	8 Aug 2002	1 Jun 2006
	GNU Emacs	200	Weekly	8 Aug 2002	1 Jun 2006
	FileZilla	200	Weekly	8 Aug 2002	1 Jun 2006
SLR 21	FreeCol	1,087	Revision	Apr 2004	Mar 2007
	GannttProject	2,701	Revision	May 2003	Dec 2006
	JBoss	5,225	Revision	Apr 2000	Jul 2006
	jEdit	1,381	Revision	Sep 2001	Jul 2006
SLR 27	ArgoUML	5,525	Revision	1 Dec 2000	7 Dec 2005
	JBoss	28,475	Revision	18 Oct 2003	N/A
	OpenSSH	1,315	Revision	26 Sep 2001	N/A
	PostgreSQL	9,324	Revision	9 Jun 1999	N/A
SLR 5	Apache Mina	22	Release	1 Oct 2006	2 Oct 2007
	jEdit	50	Release	3 Dec 2000	14 May 2004
SLR 10	Apache Ant	175	Weekly	Nov 2005	Apr 2009
	Apache httpd	200	Weekly	Jun 2005	Apr 2009
	ArgoUML	200	Weekly	Jun 2005	Apr 2009
	FileZilla	200	Weekly	Jun 2005	Apr 2009
	GIMP	200	Weekly	Jun 2005	Apr 2009
	JabRef	200	Weekly	Jun 2005	Apr 2009
	KMail	200	Weekly	Jun 2005	Apr 2009
	Nautilus	200	Weekly	Jun 2005	Apr 2009
	Umbrello	200	Weekly	Jun 2005	Apr 2009
SLR 22	Columba	3108	Revision	N/A	+52 months
	FreeCol	1087	Revision	N/A	+35 months
	GannttProject	2701	Revision	N/A	+43 months
	JBoss	3346	Revision	N/A	+30 months
	jEdit	1381	Revision	N/A	+58 months
SLR 11	Bauhaus	66	Weekly	1 Jan 2008	1 Apr 2009
SLR 26	CAROL	10	Release	12 Nov 2002	13 Apr 2005
	Claws Mail	47	Release	19 Mar 2005	31 Jun 2010
	Conky	70	Release	20 Jul 2005	30 Mar 2010
	JUnit	20	Release	12 May 2004	8 Dec 2009
	JabRef	33	Release	30 Nov 2003	14 Apr 2010
	KeePass	35	Release	17 Nov 2003	14 Oct 2006
	Nant	22	Release	25 Nov 2003	8 Dec 2007
	Notepad++	30	Release	25 Nov 2003	4 Feb 2007
	Process Hacker	38	Release	17 Oct 2008	23 Jan 2010
	Wget	17	Release	23 Sep 1998	22 Sep 2009
	ZABBIX	28	Release	23 Mar 2004	27 Jan 2010
	ZedGraph	28	Release	2 Aug 2004	12 Dec 2008
	dnsjava	22	Release	29 Mar 2001	21 Nov 2009
	eMule	73	Release	7 Jul 2002	7 Apr 2010
	iText	49	Release	7 Mar 2002	25 Jan 2008
	iTextSharp	26	Release	4 Feb 2003	8 Mar 2007
	7-Zip	45	Release	11 Dec 2003	3 Feb 2009

new relationships between the clones in the Windows Research Kernel via inspection of clustered clone groups. For example, in the memory management directory of the kernel, they found three different methods of obtaining a base address and region size, where the methods were split into

DOI: 10.1002/smr

three different clone groups due to structural differences, but were clustered together due to using several of the same words (i.e., source code identifiers).

We believe that it would be interesting to study clone lineages using semantic metrics. That is, after building clone lineages using a clone detector such as *iClones*, latent semantic indexing or latent Dirichlet allocation could be used to cluster clone lineages based on the identifiers in the corresponding clones or clone groups. The goal would be to discover latent relationships between clone lineages (or clone evolution patterns). It may even be possible to use more advanced topic modeling techniques such as hidden Markov model latent Dirichlet allocation (HMM-LDA) [39], which considers syntax (structure/order) as well as semantics (meaning), or the relational topic model [40], which identifies links between documents (clone lineages) as well as grouping words into topics.

5. CONCLUSION

Studies of clone evolution serve a key role in understanding and addressing issues of cloning in software. In this paper, we described a systematic literature review that we conducted to investigate the current state of knowledge about clone evolution. We identified 30 primary studies in accordance with our review protocol and analyzed those papers to answer three research questions. We described the methods that researchers have used to study clone evolution, the patterns that researchers have found evolving clones to exhibit, and the evidence that researchers have established regarding the extent of inconsistent change undergone by clones during software evolution.

We observed that some researchers have drawn conclusions about developer behavior or intent based only on analytical data (resulting from source code analysis). Thus, we have identified human-based empirical studies of clone evolution to be a high-priority area for future research. In addition, like others — e.g., (SLR 11, 16, 22, 26) — we observed that results of different clone evolution studies are not always consistent. However, study settings differ substantially, so we have proposed studies, which control for internal factors. Finally, Gode (SLR 10) and Koschke (SLR 11) have concluded based on their results that clone evolution patterns might be a system-specific classifier. It follows that there might be source code metrics, which can be used to predict these patterns, and we suggested both investigations using traditional source code metrics (e.g., computed on flow graphs or dependence graphs) and investigations using semantic metrics (e.g., based on topic modeling).

APPENDIX A: CLONE DETECTORS

In this appendix, we provide a list of the clone detectors that were used to perform the case studies reported in the primary studies for our systematic literature review. Note that some case studies do not focus exclusively on clone evolution, and in such cases, we list only the clone detector(s) used specifically to study clone evolution. Table V lists each clone detector, the category of its detection technique, a pointer to more information about it, and the list of primary studies in which it was used. The four categories into which we place the clone detectors are defined according to the source code representation used for detection and are: text, token, syntax, and semantics.

[4] We categorize this clone detector as semantics based, rather than metrics based, because the metrics used in the detection process are computed based on semantic information encoded in control flow graphs and data flow graphs.

APPENDIX B: SUBJECT SOFTWARE SYSTEMS

In this appendix, we list the 58 software systems that were subjects of the case studies reported in the primary studies for our systematic literature review. Table VI lists each open source software system, its implementation language, its URL, and the list of primary studies in which it was used as the subject of a case study. Table VII lists each commercial or student-authored software system, its implementation language, the providing organization or institution, and the list of primary studies in which it was used as the subject of a case study.

DOI: 10.1002/smr

Table V. The 14 clone detectors used to perform case studies.

Clone detector	Category	More information	Primary studies
Bauhaus ccdiml	Syntax	URL www.bauhaus-stuttgart.de/clones	(SLR 27)
CCFinder	Token	[41]	(SLR 8, 15, 21, 22, 23, 27–30)
CCFinderX	Token	URL ccfinder.net	(SLR 9, 26)
CP-Miner	Token	[42]	(SLR 20)
Clever	Syntax	—	(SLR 24)
CloneInspector	Token	URL wwwbroy.in.tum.de/ ccsm/icse09	(SLR 13)
Columbus	Syntax	URL frontendart.com	(SLR 3)
DECKARD	Syntax	[43]	(SLR 7, 25)
Datrix	Semantics [4]	[44]	(SLR 19)
SimScan	Syntax	URL blue-edge.bg/simscan	(SLR 2, 5, 7, 27)
Simian	Text	URL redhillconsulting.com.au/products/simian	(SLR 7, 16, 17, 18)
SmallDude	Text	URL moosetechnology.org/tools/smalldude	(SLR 4)
Untitled	Semantics [4]	[44]	(SLR 1)
iClones	Token	[45]	(SLR 10, 11)

APPENDIX C: PRIMARY STUDIES

SLR1. Antoniol G, Villano U, Merlo E, Penta MD. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology* 2002; **44**(13):755–765, doi:10.1016/S0950-5849(02)00123-4.

SLR2. Aversano L, Cerulo L, Penta MD. How clones are maintained: an empirical study. *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007; 81–90, doi:10.1109/CSMR.2007.26.

SLR3. Bakota T, Ferenc R, Gyimothy T. Clone smells in software evolution. *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, 2007; 24–33, doi:10.1109/ICSM.2007.4362615.

SLR4. Balint M, Girba T, Marinescu R. How developers copy. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006; 56–68, doi:10.1109/ICPC.2006.25.

SLR5. Bettenburg N, Shang W, Ibrahim W, Adams B, Zou Y, Hassan A. An empirical study on inconsistent changes to code clones at release level. *Proceedings of the 16th Working Conference on Reverse Engineering*, 2009; 85–94, doi:10.1109/WCRE.2009.51.

SLR6. de Wit M, Zaidman A, van Deursen A. Managing code clones using dynamic change tracking and resolution. *Proceedings of the 25th IEEE International Conference on Software Maintenance*, 2009; 169–178, doi:10.1109/ICSM.2009.5306336.

SLR7. Duala-Ekoko E, Robillard M. Clone region descriptors: representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology* Jul 2010; **20**(1):3:1–3:31, doi:10.1145/1767751.1767754.

SLR8. Geiger R, Fluri B, Gall H, Pinzger M. Relation of code clones and change couplings. *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science*, vol. 3922. Springer Berlin/Heidelberg, 2006; 411–425, doi:10.1007/11693017_31.

SLR9. German D, Di Penta M, Gueheneuc YG, Antoniol G. Code siblings: Technical and legal implications of copying code between applications. *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009; 81–90, doi:10.1109/MSR.2009.5069483.

SLR10. Göde N. Evolution of type-1 clones. *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009; 77–86, doi:10.1109/SCAM.2009.17.

SLR11. Göde N, Koschke R. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice* 2010; doi:10.1002/smr.520.

SLR12. Hou D, Jablonski P, Jacob F. CnP: towards an environment for the proactive management of copy-and-paste programming. *Proceedings of the 17th IEEE International Conference on Program Comprehension*, 2009; 238–242, doi:10.1109/ICPC.2009.5090049.

SLR13. Jürgens E, Deissenboeck F, Hummel B, Wagner S. Do code clones matter? *Proceedings of the 31st International Conference on Software Engineering*, 2009; 485–495, doi:10.1109/ICSE.2009.5070547.

Table VI. The 51 open-source software systems used as subjects of case studies.

System	Language	URL	Primary studies
Apache Ant	Java	URL ant.apache.org	(SLR 4, 10)
Apache Mina	Java	URL mina.apache.org	(SLR 5)
Apache httpd	C	URL httpd.apache.org	(SLR 10, 18, 25)
ArgoUML	Java	URL argouml.tigris.org	(SLR 2, 4, 10, 11, 16, 17, 18, 27)
CAROL	Java	URL carol.objectweb.org	(SLR 15, 16, 26)
Claws Mail	C	URL claws-mail.org	(SLR 26)
Columba	Java	URL sourceforge.net/projects/columba	(SLR 22, 24)
Conky	C	URL conky.sourceforge.net	(SLR 26)
DrJava	Java	URL drjava.org	(SLR 7)
Eclipse PDE	Java	URL eclipse.org	(SLR 8)
Eclipse JDT Core	Java	URL eclipse.org	(SLR 16, 17)
Evolution	C	URL projects.gnome.org/evolution	(SLR 25)
FileZilla	C++	URL filezilla.sourceforge.net	(SLR 10, 16, 17)
FreeBSD	C	URL freebsd.org	(SLR 9, 20, 30)
FreeCol	Java	URL freecol.org	(SLR 21, 22)
GEclipse	Java	URL eclipse.org/geclipse	(SLR 24)
GIMP	C	URL gimp.org	(SLR 10, 25)
GNU Emacs	C	URL gnu.org/software/emacs	(SLR 16, 17)
GanttProject	Java	URL ganttproject.biz	(SLR 21, 22)
JBoss	Java	URL jboss.org	(SLR 21, 22, 27)
JMeter	Java	URL jakarta.apache.org/jmeter	(SLR 7)
JUnit	Java	URL junit.org	(SLR 26)
JabRef	Java	URL jabref.sourceforge.net	(SLR 10, 26)
KMail	C++	URL userbase.kde.org/Kmail	(SLR 10)
KeePass	C++	URL keepass.info	(SLR 26)
Linux Kernel	C	URL kernel.org	(SLR 1, 9, 20)
Mozilla	C++	URL www-archive.mozilla.org/releases	(SLR 8)
Mozilla Firefox	C++	URL mozilla.com/firefox	(SLR 3)
Nant	C#	URL nant.sourceforge.net	(SLR 26)
Nautilus	C	URL projects.gnome.org/nautilus	(SLR 10, 25)
NetBSD	C	URL netbsd.org	(SLR 30)
Notepad++	C++	URL notepad-plus-plus.org	(SLR 26)
OpenBSD	C	URL openbsd.org	(SLR 9, 30)
OpenSSH	C	URL openssh.com	(SLR 27)
PostgreSQL	C	URL postgresql.org	(SLR 27)
Process Hacker	C#	URL processhacker.sourceforge.net	(SLR 26)
Ptolemy II	Java	URL ptolemy.berkeley.edu/ptolemyII	(SLR 4)
SQuirreL	Java	URL squirrel-sql.sourceforge.net	(SLR 17)
Sysiphus	Java	URL sysiphus.in.tum.de	(SLR 13)
Umbrello	C++	URL umbrello.org	(SLR 10)
Wget	C	URL gnu.org/software/wget	(SLR 26)
ZABBIX	C	URL zabbix.com	(SLR 26)
ZedGraph	C#	URL zedgraph.org	(SLR 26)
dnsjava	Java	URL dnsjava.org	(SLR 2, 15, 23, 26)
eMule	C++	URL emule-project.net	(SLR 26)
gcc	C	URL gcc.gnu.org	(SLR 11)
iText	Java	URL itextpdf.com	(SLR 26)
iTextSharp	C#	URL sourceforge.net/projects/itextsharp	(SLR 26)
jEdit	Java	URL jedit.org	(SLR 5, 7, 21, 22, 24)
4.4BSD	C	URL freebsd.org	(SLR 30)
7-Zip	C++	URL 7-zip.org	(SLR 26)

SLR14. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. *Proceedings of the 3rd International Symposium on Empirical Software Engineering*, 2004; 83–92, doi:10.1109/ISESE.2004.1334896.

SLR15. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005; 187–196, doi:10.1145/1081706.1081737.

DOI: 10.1002/smr

Table VII. The commercial-authored or student-authored software systems used as subjects of case studies.

System (#)	Language	Organization	Primary studies
Bauhaus	Ada	Axivion	(SLR 11)
Commercial (x3)	C#	Munich Re	(SLR 13)
Commercial	Cobol	LV 1871	(SLR 13)
Commercial	Pascal-like	*Telecommunications*	(SLR 19)
Student	C	Osaka University	(SLR 28, 29)

SLR16. Krinke J. A study of consistent and inconsistent changes to code clones. *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007; 170–178, doi:10.1109/WCRE.2007.7.
SLR17. Krinke J. Is cloned code more stable than non-cloned code? *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008; 57–66, doi:10.1109/SCAM.2008.14.
SLR18. Krinke J, Gold N, Jia Y, Binkley D. Distinguishing copies from originals in software clones. *Proceedings of the 4th International Workshop on Software Clones*, 2010; 41–48.
SLR19. Lagüe B, Proulx D, Mayrand J, Merlo E, Hudepohl J. Assessing the benefits of incorporating function clone detection in a development process. *Proceedings of the 13th IEEE International Conference on Software Maintenance*, 1997; 314–321, doi:10.1109/ICSM.1997.624264.
SLR20. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* Mar 2006; **32**(3):176–192, doi:10.1109/TSE.2006.28.
SLR21. Lozano A, Wermelinger M. Assessing the effect of clones on changeability. *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008; 227–236, doi:10.1109/ICSM.2008.4658071.
SLR22. Lozano A, Wermelinger M. Tracking clones' imprint. *Proceedings of the 4th International Workshop on Software Clones*, 2010; 65–72.
SLR23. Lozano A, Wermelinger M, Nuseibeh B. Evaluating the harmfulness of cloning: a change based experiment. *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007; 18:1–18:4.
SLR24. Nguyen T, Nguyen H, Pham N, Al-Kofahi J, Nguyen T. Clone-aware configuration management. *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009; 123–134, doi:10.1109/ASE.2009.90.
SLR25. Rahman F, Bird C, Devanbu P. Clones: what is that smell? *Proceedings of the 7th Working Conference on Mining Software Repositories*, 2010; 72–81, doi:10.1109/MSR.2010.5463343.
SLR26. Saha R, Asaduzzaman M, Zibran M, Roy C, Schneider K. Evaluating code clone genealogies at release level: an empirical study. *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2010; 87–96, doi:10.1109/SCAM.2010.32.
SLR27. Thummalapenta S, Cerulo L, Aversano L, Penta MD. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* Mar 2009; **15**(1):1–34, doi:10.1007/s10664-009-9108-x.
SLR28. Ueda Y, Kamiya T, Kusumoto S, Inoue K. Gemini: maintenance support environment based on code clone analysis. *Proceedings of the 8th IEEE International Symposium on Software Metrics*, 2002; 67–76, doi:10.1109/METRIC. 2002.1011326.
SLR29. Ueda Y, Kamiya T, Kusumoto S, Inoue K. On detection of gapped code clones using gap locations. *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002; 327–336, doi:10.1109/APSEC.2002.1183002.
SLR30. Yamamoto T, Matsushita M, Kamiya T, Inoue K. Similarity of software system and its measurement tool SMMT. *Systems and Computers in Japan* Jun 2007; **38**(6):91–99, doi:10.1002/scj.10379.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments and helpful suggestions. The work presented in this article was supported in part by the National Science Foundation under Grant Nos. 0851824 & 0915559. The authors gratefully acknowledge the NSF for its support.

REFERENCES

1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code* (1st edn). Addison-Wesley: Upper Saddle River, NJ, USA, 2000.
2. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005; 187–196, doi:10.1145/1081706.1081737.
3. Thummalapenta S, Cerulo L, Aversano L, Penta MD. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 2009; **15**(1):1–34, doi:10.1007/s10664-009-9108-x.
4. Kapser C, Godfrey M. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering* 2008; **13**(6):645–692, doi:10.1007/s10664-008-9076-6.
5. Rahman F, Bird C, Devanbu P. Clones: what is that smell? *Proceedings of the 7th Working Conference on Mining Software Repositories*, 2010; 72–81, doi:10.1109/MSR.2010.5463343.
6. Göde N, Koschke R. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice* 2010; doi:10.1002/smr.520.
7. Koschke R. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings, 2007. http://drops.dagstuhl.de/opus/volltexte/2007/962.
8. Roy C, Cordy J. A survey on software clone detection research. *Technical Report 2007–541*, School of Computing, Queen's University, Kingston, Ontario, Canada Sep 2007. http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf.
9. Harder J, Göde N. Modeling clone evolution. *Proceedings of the 3rd International Workshop on Software Clones*, 2009.
10. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. *Proceedings of the 3rd International Symposium on Empirical Software Engineering*, 2004; 83–92, doi:10.1109/ISESE.2004.1334896.
11. Göde N, Koschke R. Frequency and risks of changes to clones. *Proceedings of the International Conference on Software Engineering*, 2011; 311–320.
12. Biolchini J, Mian P, Natali A, Travassos G. Systematic review in software engineering. *Technical Report ES 679/05*, PESC-COPPE/UFRJ, Brazil May 2005. http://www.cos.ufrj.br/uploadfiles/es67905.pdf.
13. Kitchenham B, Charters S. Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report EBSE-2007-01*, School of Computer Science and Mathematics, Keele University and Department of Computer Science, University of Durham Jul 2007. http://www.elsevier.com/framework products / promismisc / inf-systrev.pdf.
14. Duala-Ekoko E, Robillard M. Clone region descriptors: representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology* 2010; **20**(1):3:1–3:31, doi:10.1145/1767751.1767754.
15. Kapser C, Anderson P, Godfrey M, Koschke R, Rieger M, van Rysselberghe F, Weisgerber P. Subjectivity in clone judgment: can we ever agree? *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings, 2007. http://drops.dagstuhl.de/opus/volltexte/2007/970.
16. Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 2007; **33**(9):577–591, doi:10.1109/TSE.2007.70725.
17. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Science of Computer Programming* 2009; **74**(7):470–495, doi:10.1016/j.scico.2009.02.007.
18. Bettenburg N, Shang W, Ibrahim W, Adams B, Zou Y, Hassan A. An empirical study on inconsistent changes to code clones at release level. *Proceedings of the 16th Working Conference on Reverse Engineering*, 2009; 85–94, doi:10.1109/WCRE.2009.51.
19. Hannay J, Sjoberg D, Dyba T. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering* 2007; **33**(2):87–107.
20. Jorgensen M, Shepperd M. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering* 2007; **33**(1):33–53.
21. Walia G, Carver J. A systematic literature review to identify and classify software requirement errors. *Information and Software Technology* 2009; **51**(7):1087–1109.
22. Williams B, Carver J. Characterizing software architecture changes: a systematic review. *Information and Software Technology* 2010; **52**(1):31–51.
23. Mann Z. Three public enemies: cut, copy, and paste. *Computer* 2006; **39**(7):31–35.
24. Toomim M, Begel A, Graham S. Managing duplicated code with linked editing. *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, 2004; 173–180.
25. Bevan J, Whitehead, Jr E, Kim S, Godfrey M. Facilitating software evolution research with Kenyon. *SIGSOFT Software Engineering Notes* 2005; **30**(5):177–186, doi: http://doi.acm.org/10.1145/1095430.1081736.
26. Canfora G, Cerulo L, Penta MD. Identifying changed source code lines from version repositories. *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007; 14.
27. Levenshtein V. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics – Doklady* 1966; **10**(8):707–710.
28. Cordy J. Comprehending reality — practical barriers to industrial adoption of software maintenance automation. *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IEEE Computer Society: Washington, DC, USA, 2003; 196.
29. Chatterji D, Carver J, Massengill B, Oslin J, Kraft N. Measuring the efficacy of code clone information in a bug localization task: an empirical study. *Proceeding of the 5th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2011.
30. Carver J, Nagappan N, Page A. The impact of educational background on the effectiveness of requirements inspections: an empirical study. *IEEE Transactions on Software Engineering* 2008; **34**(6):800–812.

DOI: 10.1002/smr

31. Carver J, Chatterji D, Kraft N. On the need for human-based empirical validation of techniques and tools for code clone analysis. *Proceeding of the 5th International Workshop on Software Clones*, 2011; 61–62.
32. Harder J, Göde N. Efficiently handling clone data: RCF and cyclone. *Proceeding of the 5th International Workshop on Software Clones*, 2011.
33. Ãšjházi B, Ferenc R, Poshyvanyk D, Gyimóthy T. New conceptual coupling and cohesion metrics for object-oriented systems. *Proceedings of 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2010; 33–42.
34. Deerwester S, Dumais S, Landauer T, Furnas G, Harshman R. Indexing by latent semantic analysis. *Journal of the Society for Information Science* 1990; **41**(6):391–407.
35. Blei D, Ng A, Jordan M. Latent Dirichlet allocation. *Journal of Machine Learning Research* 2003; **3**:993–1022.
36. Kuhn A, Ducasse S, Girba T. Semantic clustering: identifying topics in source code. *Information and Software Technology* 2007; **49**:230–243.
37. Tairas R, Gray J. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering* 2009; **14**(1), doi: 10.1007/s10664-008-9089-1.
38. Marcus A, Maletic J. Identification of high-level concept clones in source code. *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001; 107–114.
39. Griffiths T, Steyvers M, Blei D, Tenenbaum J. Integrating topics and syntax. *Advances in Neural Information Processing Systems 17*, 2005.
40. Chang J, Blei D. Hierarchical relational models for document networks. *Annals of Applied Statistics* 2010; **4**(1):124–150.
41. Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
42. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004; 289–302.
43. Jiang L, Misherghi G, Su Z, Glondu S. DECKARD: scalable and accurate tree-based detection of code clones. *Proceedings of the 29th International Conference on Software Engineering*, 2007; 96–105.
44. Mayrand J, Leblanc C, Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. *Proceedings of the 12th International Conference on Software Maintenance, Monterey, CA, USA*, IEEE Computer Society, 1996; 244–253.
45. Göde N, Koschke R. Incremental clone detection. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, 2009; 219–228.

AUTHORS' BIOGRAPHIES:

Jeremy R. Pate received the BSc degree in computer science from The University of Alabama in 2008. He is currently a PhD student in the Department of Computer Science and is a programmer/analyst for the Center for Advanced Public Safety, both at The University of Alabama. His research interests include domain-specific languages, reverse engineering, and clone evolution.

Robert Tairas received the BSc degree in computer science and mathematics from Samford University in 1997. He received the MS and PhD degrees in computer and information sciences from The University of Alabama at Birmingham in 2005 and 2010, respectively. He is currently a postdoctoral research fellow with the AtlanMod team, an INRIA research group at École des Mines de Nantes (France). His research interests include code clones and model-driven engineering.

Nicholas A. Kraft received the BA degree in mathematics from Indiana University in 2002 and the PhD degree in computer science from Clemson University in 2007. He is currently an assistant professor in the Department of Computer Science at The University of Alabama. His research interests include program comprehension, reverse engineering, grammar engineering, and software maintenance and evolution. He is a member of the ACM and the IEEE.

DOI: 10.1002/smr