Computing and Informatics, Vol. 40, 2021, 690–728, doi: 10.31577/cai_2021_3_690

WHAT IS YOUR CODE CLONE DETECTION AND EVOLUTION RESEARCH MADE OF?

Chaman Wijesiriwardana

University of Moratuwa, Katubedda
Sri Lanka
e-mail: chaman@uom.lk

Prasad Wimalaratne

University of Colombo School of Computing
Reid Avenue, Colombo 7, Sri Lanka
e-mail: spw@ucsc.cmb.ac.lk

Abstract. Over the past few decades, clone detection and evolution have become a major area of study in software engineering. Clone detection experiments present several challenges to researchers such as accurate data collection, selecting proper code detection algorithms, and understanding clone evolution phenomena. This paper attempts to facilitate clone detection and evolution research by providing a structured and systematic mechanism to conduct experiments. Clone detection experiments usually consist of several tasks such as fetching data from a version control system, performing necessary pre-processing activities, and feeding the data to a clone detection algorithm. Therefore, a particular clone detection experiment can interpret as a meaningful combination of such tasks into a scientific workflow. In this work, the concrete tasks in a code clone detection workflow are referred to as Building Blocks. This paper presents a useful collection of Building Blocks identified based on a systematic literature review, and a conceptual framework of an experimental testbed to facilitate clone detection experiments. The reusability of the Building Blocks was validated using four case studies selected from the literature. The validation results confirm the reusability and the expressiveness of the Building Blocks in new ventures. Besides, the proposed experimental testbed is proven beneficial in conducting and replicating clone detection experiments.

Keywords: Code clone detection, clone evolution, scientific workflows, building blocks, experiemental testbed

1 INTRODUCTION

Code clones are source code fragments that are similar or identical in terms of text, structure, or meaning. During software development, code fragments are copied and pasted with or without major alternations. The pasted portion of the code is said to be a clone, and this practice is known as code cloning. This has been a common practice in the software development process due to several reasons such as limitations of the programming languages, delaying refactoring, and high code reuse [46]. The negative impacts of code cloning has been reported in several studies [34, 3, 8]. The consequences of code cloning, clone evolution, and clone removal have both positives and negatives. Fowler et al. [24] were one of the firsts to argue that code cloning is one of the leading causes of *bad smells* in software systems. On the other hand, some researchers counter-argued by highlighting several positives of cloning such as improved productivity [5, 43, 71]. According to [15], there can be organizational reasons to copy-paste code. Therefore, a systematic analysis is required before the clone removal. As a result, clone removal can sometimes directly associate with a considerable risk factor. More research along this direction should be conducted to get a better understanding, and the uncertainties mentioned above evidently emphasize the need for systematic Code Clone Detection and Evolution (CCDE) experiments.

This paper contributes to this field of study by proposing a structured and organized way to plan and conduct CCDE experiments. Clone detection studies are conducted based on a well-defined logical process with some common steps. For example, studies typically start with mining activity, e.g., by retrieving data from a version control system. Then the mined source code is processed and transformed into an intermediate format, such as tokens, code metrics, Abstract Syntax Trees (ASTs), or Program Dependency Graphs (PDGs). This data is then fed into a code clone detection algorithm to extract the clones. Finally, the detected code clones can be further subjected to clone genealogy analysis (i.e., clone patterns, visualization) to understand clone evolution better. This research insists that the steps mentioned above can adequately arrange and combine into a well-defined scientific workflow [18, 80]. Each step represents a particular clone detection task, such as mining the code base, calculating metrics, or generating an AST. These tasks may be in different representation levels and the effort required to perform the tasks may have significant differences. However, the identification of representation levels and the effort required for the tasks are out of the scope of this paper. In this paper, these concrete tasks are referred to as *Building blocks* for CCDE research. We believe that the concept of *Building blocks* will provide direct solutions to some of the common challenges such as accurate data extraction, data cleaning, and pick the

correct clone detection technique in conducting CCDE studies. Therefore, *Building blocks* provide ways to thoroughly comprehend the clone detection process as well as to replicate previous experiments systematically. This work devised a conceptual framework and a proof-of-concept experimental testbed to conduct CCDE experiments, which could be extended to conduct other types of software engineering experiments as well.

This paper intends to address the following Research Questions (RQs):

RQ1: Is it possible to identify reusable data flow-based *Building Blocks* for code clone detection and evolution from the existing literature and express them in a unified manner?

RQ2: How to interlink such *Building Blocks* categorized into multiple abstraction levels to develop a conceptual framework of an experimental testbed to conduct code clone detection and evolution experiments?

Based on the research questions, this paper presents three main contributions. First, a novel concept and a methodology for identifying re-usable building blocks from various research workflows in the area of code clone detection. Second, a concrete collection of useful formal building blocks was collected via the above methodology. Finally, a conceptual framework of an experimental testbed by interlinking the identified building blocks to conduct and replicate previous CCDE experiments.

The remainder of this paper is organized as follows: Section 2 describes the background of software evolution analysis and code clone detection. Section 3 details our research methodology to identify building blocks from the analysis workflows. In Section 4, we present our catalog of building blocks extracted via the literature survey followed by the conceptual framework of the experimental testbed in Section 5. Then we present validation in Section 6 followed by the discussion and conclusion in Section 7 and Section 8, respectively.

2 BACKGROUND

Analysis of software evolution is known as an enormously dynamic field of research in software engineering. Understanding the evolution of large-scale software systems is a demanding problem for several reasons: huge amounts of information have to be considered, and historical data has to be analyzed. Software evolution analysis mainly focuses on two main aspects; to better understand the reasons for its existing problems and to forecast its future developments [17]. Software evolution analysis experiment such as CCDE is a classic example to address both of these goals. First, we summarize some of the pioneer surveys in CCDE and indicate how our approach conceptually differs from the existing surveys. Then we dig into the CCD techniques and tools followed by a summary of the existing approaches for software engineering data analytics. Finally, we briefly describe the need for a novel mechanism to facilitate the researchers in conducting CCDE experiments.

2.1 Surveys in Code Clone Detection and Evolution

A considerable number of code clone related survey papers are published in the past. Koshke [46] was one of the first to write a survey paper on code clone detection. That paper reports some essential aspects such as different categorizations of clone types, root causes for cloning, current opinions of cloning, empirical studies on the evolution of clones, benchmarks for clone detector evaluations, and presentation issues.

Roy et al. [70] presented a qualitative comparison and evaluation of the existing literature in clone detection techniques and tools. A more detailed description can be found in [68]. The findings of their research could help new potential users of clone detection techniques in understanding the range of available techniques and tools and selecting those most appropriate for their needs. Ratten et al. [67] perform a systematic review of existing code clone approaches based on 213 identified papers. The results are presented in different dimensions like classification of clone research, code clone management as cross-cutting domain, types of clones, clone detection tools, and clone detection approaches. Similar to [68], this approach also intends to facilitate researchers in conducting code clone detection researches. Sheneamer et al., [76] also surveyed code clone detection. The aim of this paper goes beyond comparing the tools and techniques. Instead, it presents several observations in developing hybrid techniques in the future. Pate et al. present a survey paper in code clone evolution., [63]. They have indicated that human-based empirical studies and classification of clone evolution patterns as two significant areas for further work. Ain et al. [3] reviewed 54 journal papers and conference papers, which emphasized the need to introduce novel approaches to detect all four types of clones. Walker et al. [84] presented a systematic mapping study on existing CCD tools with regards to technique, open-source nature, and language coverage. Finally, they propose some possible future directions for code-clone detection tools.

2.2 Code Clone Detection: Techniques and Tools

Code reuse is a frequent activity in software development. Code reusing can be in the form of copying a portion of the code and pasting it with or without modifications. This type of reuse known as code cloning and the pasted code fragment is called a clone of the original code [70]. Nevertheless, during the maintenance stage, identifying the original code fragment and the copied code fragment is a non-trivial task. Several clone detection approaches have been proposed in the literature spanning from textual to semantic approaches. However, this paper does not consider the techniques and tools proposed for cross-language clone detection [57].

Text based clone detection: During this approach, code fragments are compared with each other in the form of texts; strings or lexemes and similar portions are identified as code clones [67]. One of the earliest clone detection approaches was proposed by Johnson [36, 38]. He applied a fingerprinting mechanism for

comparison of source code. Ducasse et al. [21] developed a language independent clone detection tool, duploc, which aims at overcoming the obstacle of having the right parser for the right dialect for every language. However, this approach requires a significant effort in pre-processing and transforming the source code into the required syntax. Duploc cannot detect Type-3 clones or deal with modifications and insertions in copy-pasted code [76]. NICAD [69] is a text-based clone detection tool that is capable of effectively detecting clones up to type 3. It is based on lightweight parsing to implement code normalization and code filtering. Seunghak and Jeong [49] presented a text-based code clone detection technique. They have implemented Similar Data Detection (SDD) tool, which is an Eclipse plug-in. Dou et al. [20] has effectively used text-based clone detection technique in detecting clones in spreadsheets.

Token based clone detection: Token-based clone detection techniques are considered better than text-based detection. In this approach, lexical analysis is used to extract the tokens from the source code by lexical analysis. One of the focal points behind token-based clone detection algorithms is to perform suffix tree or suffix array based token-to-token comparisons. A suffix tree is a data structure that exposes the internal structure of a string in a deeper way [28]. Suffix array is also a conceptually simple data structure which is initially developed for on-line string searches [54]. The central advantage of suffix arrays over suffix trees is that, in practice, suffix arrays use three to five times less space. CCFinder [40] is a well-known tool of this category, which finds identical subsequences by using a suffix tree matching algorithm. The research community for code clone analysis as well as code clone management broadly uses CCFinder. Dup [7, 6] is another token-based clone detection tool, which divides the source files into tokens by a lexical analyzer. CCLEARNER [51] is a token based clone detection tool developed by leveraging deep learning. However, approaches such as [69] and [74] are not exploiting suffix trees or arrays in detecting code clones. For example, Sajanai et al., [74] uses an optimized partial index and filtering heuristics to achieve large-scale clone detection. This technique has used in recent studies as well [73].

Tree-based clone detection: In tree-based clone detection algorithms, the source code transformed into a parse tree or an abstract syntax tree. A parse tree is a data structure for the parsed representation of a statement in a particular code fragment [12]. The usefulness of generating the parse tree is that, by parsing two code fragments and comparing their parse trees, it is possible to determine whether the code fragments are identical or not. Abstract Syntax Tree is also a special kind of parse tree. In parse trees, the roots of subtrees represent nonterminal symbols of the grammar, while leaves represent terminal grammar symbols. In an abstract syntax tree, operators represent root nodes, while leaves symbolize operands [60]. Once the trees are generated, tree-matching algorithms are used to find the similar code fragments. One of the first approaches of this category was presented by Yang [89]. CloneDR [11] is another token-based clone

detection tool, which can detect exact and near-miss clones using hashing and dynamic programming. Wahler et al. [83] presented a technique to find clones at a more abstract level by converting abstract syntax tree to XML format.

Program dependency graph based clone detection: A program dependence graph (PDG) is a graph representation of a code fragment. In PGDs, basic statements such as variable declarations, assignments, and function calls are represented by program vertices. Edges between program vertices in PDGs represent the data and control dependencies between statements [23]. In Program Dependency Graphs (PDG), the source code is abstracted to extract the control flow and data flow graphs. Krinke [48] has presented a methodology for identifying similar code based on finding similar maximal sub-graphs by using the k-limiting technique. Hugo and Kusumoto [29] proposed a methodology to enhance PGD based clone detection based on PDG specializations and detection heuristics. Clone detection tools based on PGDs, as proposed in [44] and [87], can be used to identify type 4 clones.

Metric based clone detection: In a metric-based approach [45, 55] the source code is divided into smaller units (e.g., one line, one method, one class) and metrics are calculated for each unit. The metrics of each unit are compared, and those with the same values are identified as clones. Examples of metrics are the number of function calls within a unit or the cyclomatic complexity of the unit. The type of metrics used by each tool impacts the language dependency of the tool.

Hybrid clone detection: Hybrid clone detection techniques typically employ a combination of clone detection techniques. A hybrid approach aims at overwhelming the problems encountered by specific techniques. Leitao [50] presents a hybrid approach that combines syntactic techniques and semantic techniques with specialized comparison functions. Hummel et al. [33] present ConQat, which is an incremental index-based hybrid technique to detect clones. Agrawal et al. [2] described a hybrid approach by combining token-based and textual approaches to find code cloning. Hu et al. [31, 32] recently proposed BINMATCH, which is a hybrid approach to detect binary clone functions.

2.3 Code Clone Evolution

Software evolves from one version to another when adding new features, getting involved with fixing bugs, improving performance and increasing reliability. As a result, code clones also evolve simultaneously together with software systems. Therefore, analysis of code clone evolution is critical to comprehend the effect of code clones to the entire software system.

There are several noteworthy studies in the literature on code clone evolution with a particular focus on clone genealogies. Kim et al. [43] has pioneered one of the first investigations on code clone evolution. That paper defined clone genealogies as the history of how each element in a group of clones has changed concerning

other elements in the same group. In that research, they emphasized the need for understanding clone genealogies to maintain code clones better. Saha et al. [71] extended the research conducted by Kim et al. [43] by incorporating different dimensions. They have presented an empirical study to investigate clone genealogies using 17 open source software systems. Clone evolution related investigations have been further reported by Göde [26], Barbour et al. [8] and Krinke [47].

2.4 Software Engineering Data Analytics

Existing frameworks for software engineering data analytics based on either generic query languages such as SQL or domain-specific languages. Some of the approaches that directly follow the standard SQL syntax are Gitana [16], AlitheiaCore [27] and MetricMiner [77]. However, such approaches not specifically targeted at facilitating CCDE experiments. Thus, CCDE specific functionalities are not available. Besides, the domain-specific languages such as Boa [22] and QWALKEKO [78] requires a prior understanding of the language itself. Hence, the usability of such frameworks, particularly, for novice researchers is questionable.

2.5 Summary

Our approach is conceptually different from the previous work. Existing survey papers on CCDE mainly focused on identifying the different techniques, tools, compare them, selecting appropriate technique, and present the observations on future CCD tools. This paper presents a different dimension to facilitate researchers in understanding and conducting code clone detection experiments by utilizing a set of Building Blocks that are meant to CCD experiments. For example, conducting comparison studies is hard for the researchers as the clone detection techniques are naturally complex as there are many different pre-processing activities, transformation activities, and algorithms are involved. Therefore, a unified framework to facilitate replication studies is important, and to the best of our knowledge, such frameworks are not adequately presented in the literature. Thus, we believe that our approach would shed light on future directions such as [63].

3 APPROACH

This research aims at providing a systematic way to conduct CCDE experiments by identifying reusable tasks from the literature. Initially, a literature review on papers published on reputed software engineering conferences such as ICSE[1], MSR[2], ICSM(E)[3] and FSE[4] conducted for 11 years (2010-2020). All the papers are ex-

[1] International Conference on Software Engineering
[2] Working Conference on Mining Software Repositories
[3] International Conference on Software Maintenance (and Evolution)
[4] Foundations of Software Engineering

tracted from the main track of each of the conferences. First, the papers published on CCDE has filtered and carefully investigated the methodology section in each article. However, the study is not strictly limited to the baseline papers on the mentioned conferences in the given duration. We further traced back and forth to find related work published outside the selected papers as well. Based on this study, a mechanism has introduced to drill-down the experiments to identify the concrete, reusable tasks. From the clone detection experiments, four main *activities* has identified: data gathering, pre-processing, clone detection and post-processing. Such activities are implemented via smaller tasks or sub-tasks, which refers to as *Building Blocks* in this paper. Finally, we proposed a method to represent the building blocks using a semi-structured textual notation and a graphical notation. Below we provide an overview of our proposed methodology.

This review can be further extended by considering the other reputed software engineering conferences. However, the objective of this study is to find a useful collection of building blocks to conduct CCDE experiments. Thus, the papers published in the selected conferences were rich enough to identify the building blocks for CCDE.

3.1 Data Collection

The process of selecting suitable research publications for a particular review has two major problems: identifying the relevant work and assessing the quality of the selected studies. Therefore, it was decided to minimize the risk of errors by mainly reviewing the papers published in well-reputed software engineering conferences. A literature review is conducted in the proceedings of the ICSE, MSR, ICSM(E) and FSE conferences for eleven years (2010–2020). From that the papers published on *code clone detection* has filtered out. Table 1 presents a summary of the reviewed papers.

Year	**No. of Papers Reviewed**			
	ICSE	**MSR**	**ICSM(E)**	**FSE**
2010	1	2	3	3
2011	2	1	3	0
2012	4	0	3	0
2013	3	2	3	1
2014	3	3	7	1
2015	0	1	2	1
2016	1	0	1	1
2017	2	1	2	0
2018	1	2	2	1
2019	3	1	2	0
2020	3	0	1	0

Table 1. Number of reviewed clone detection papers

3.2 Drilling Down Code Clone Detection Experiments to Building Blocks

Scientific workflows are meant to be data flow oriented, which facilitates streamlining of scientific tasks to make significant scientific discoveries [53]. They widely recognized as a useful mechanism to describe and manage complex scientific analyses. Scientific workflows provide means to specify how a specific experiment can be modeled and carried out. In such workflows, relevant activities need to sequence in a pipeline to create a workflow that can execute a particular analysis experiment. Therefore, a specific CCDE experiment can interpret as a problem of creating a suitable workflow and running it without interruptions. In this research, scientific workflows are considered to be the top level abstraction of CCDE experiments.

Activities: A CCDE workflow is composed of activities. Activities are the tasks and sub-tasks that directly associated with CCDE experiments such as fetching data from VCS, calculating metrics or removing test files. In this paper, tasks that serve a specific analysis or perform a specific operation are grouped under a particular Activity. For example, extracting data from a VCS is an essential task in CCDE experiments. Therefore, fetch data from GIT, fetch data from SVN and fetch data from CVS can be grouped under an activity called *data gathering*. Similarly, tasks such as snapshot generation and token generation can pool under the activity called *pre-processing*. As explained in [43], though it is not necessary to follow all the steps, a typical clone detection process follows the key activities namely pre-processing, transformation, match detection, formatting, filtering, and aggregation. We slightly modified the activities while keeping the core concept unchanged to fit into our context, as shown in Figure 1.

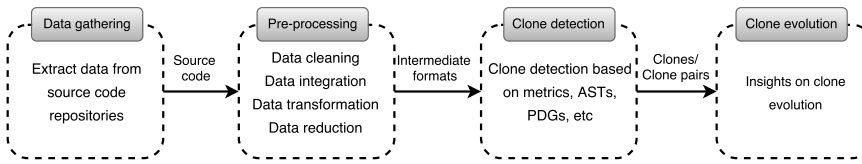

Figure 1. Common activities in a code clone detection and evolution workflow

1. Data gathering: During this phase, historical data about software projects are extracted from version control systems (e.g., CVS, SVN, GIT). Given that one of the leading contributions of this research is to identify reusable BBs in CCDE research, the other variant of data extractions such as gathering data from binaries has not considered.
2. Pre-processing: Data pre-processing can be in different forms such as data cleaning, data conversion or data integration. For example, data originating from the version control repositories need to be converted to different formats to facilitate various kinds of CCDE experiments. Furthermore, data from a VCS has to tokenize before feeding it into a token-based clone detection algorithm. The pre-processing steps can be carried out within the

clone detectors as well. However, separating it out from clone detectors has several notable advantages. For instance, novice researchers can better understand the fine-grained details of the entire CCDE process. Besides, the BBs in the pre-processing stage can be utilized in various software evolution experiments.

3. Clone detection: During this step, how the different clone detection approaches function in different settings are described. For example, after tokenizing the source code in the pre-processing stage, it has to be fed into a token-based clone detection algorithm. Similarly, metrics-based clone detectors depend on the metrics that generated in the pre-processing step.
4. Clone evolution: The primary goal of this step is to investigate how the clone detection results can utilize in clone evolution. For example, code clone genealogies provide useful insights to express how code clones change over multiple versions of the software.

Building Blocks: Activities are implemented via *Building Blocks* (BBs). A building block noticeably represents a specific analysis task such as fetch data from GIT, fetch data from SVN, generate snapshots or create ASTs. Figure 2 is a graphical representation of such BBs. Each BB is responsible only for a small fragment of functionality. Dependencies between BBs within a workflow determined by a list of parameters such as input and output parameters, pre-conditions and post-conditions. Input parameters and the pre-conditions are directly associated with Predecessor BBs, whereas output parameters and the post-conditions are directly associated with Follow-up BBs. Predecessor BBs and Follow-up BBs are two properties used to represent a particular BB, which explains in the next paragraph. For example, if an input parameter of an activity B is connected to an output parameter of activity A, it means that activity A must execute before activity B, and the data produced by activity A is consumed by activity B. The connection logic is explained in Section 6.2 under the implementation details of the proposed experimental testbed. Therefore, more comprehensively, we can infer that CCDE experiments consist of activities (e.g., data gathering, pre-processing), which are implemented via building blocks (e.g., SVN miner, AST generator).

Representing Building Blocks: A building block may consist of processes, data sources, operators and relationships. A process is a concrete example of activity, as mentioned earlier (e.g., mine version control repositories, mine bug repositories). The data source can be a source code repository such as Git, a bug repository such as Bugzilla, or any other intermediate location that keeps data. Operators are the basic operations that could perform on data (e.g., filter, sort). Likewise, a set of well-defined building blocks will be created, offering the option to use and combine such building blocks into a workflow to solve or to better understand complex CCD tasks.The usefulness and reusability of BBs mainly depend on the degree of expressiveness of such BBs. This expressiveness allows easily identifying when and why to use specific BBs, as well as when and why not

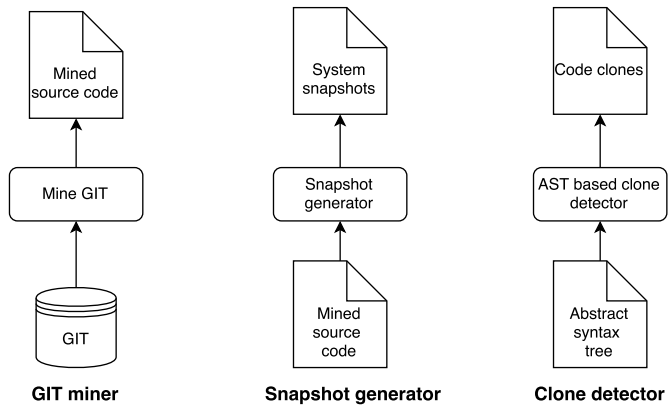

Figure 2. Building blocks in a CCDE workflow

use them. As per our understanding, there is no universally accepted standard for representing any sort of analysis tasks, in our case *Building Blocks*. However, we consider the following properties are rich enough to effectively describe a BB.

- Building block name: name of the BB
- Activity name: high level activity of the BB
- Problem(situation): why and when to apply the BB
- Solution: how to apply the BB and what it exactly does
- Alternatives: what other BBs could be used to replace this BB
- Predecessors: what other BBs should be executed prior to this BB
- Follow-up BBs: what other BBs could be executed after this BB
- References: how other researchers have used this BBs in conducting CCD experiments

4 BUILDING BLOCKS FOR CODE CLONE DETECTION AND EVOLUTION

This section presents a catalog of BBs, which classified into four main activities: data gathering, pre-processing, clone detection and post-processing. Such BBs further described by using the properties mentioned in Section 3.2. Individual BBs are not represented graphically. However, an overall view of BBs and how they can meaningfully interconnect with other BBs provided in Figure 3. This representation goes beyond a simple classification, but provide insights to the researchers on how to utilize and compose BBs to solve a particular analysis task.

Building Blocks for Data Gathering: Software evolution experiments typically require information about software projects that are collected via repository

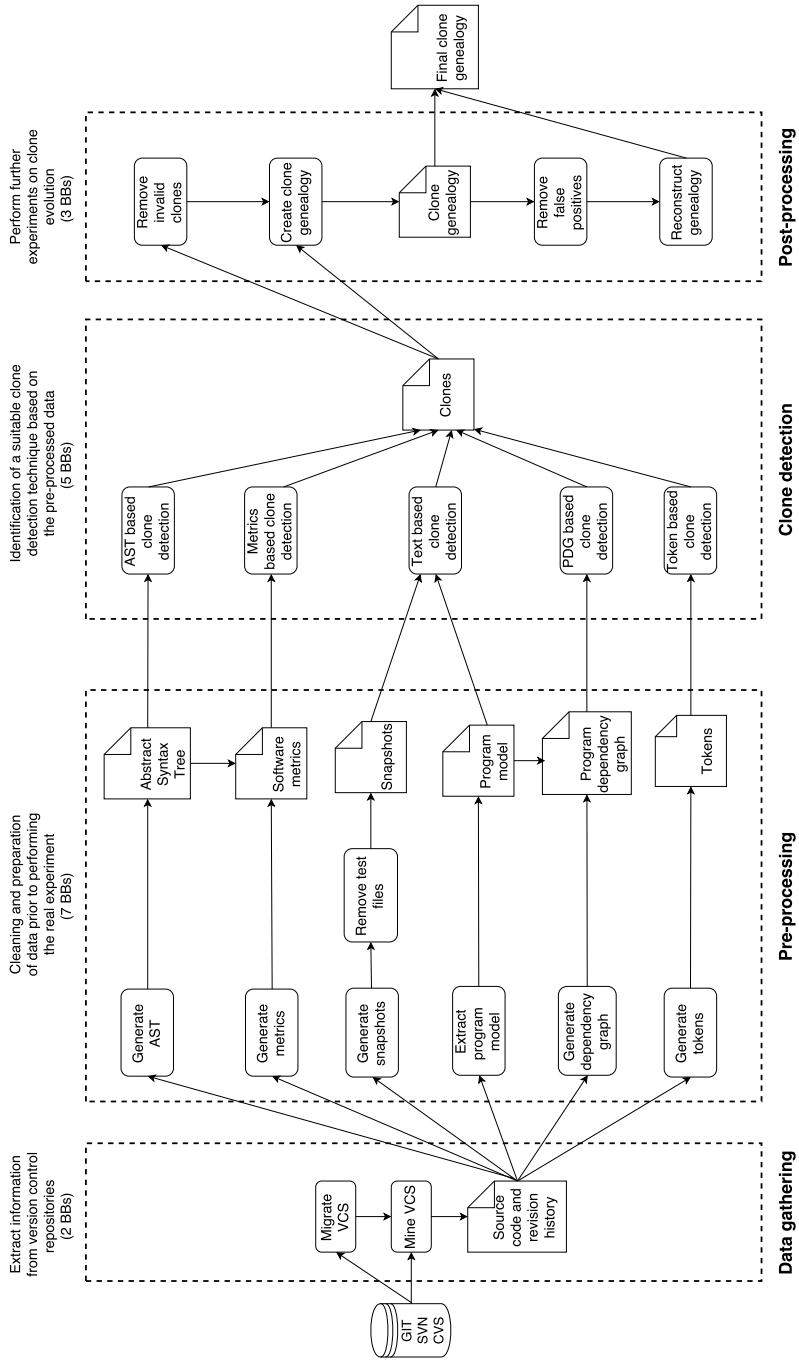

Figure 3. Activities and building blocks in code clone detection and evolution

mining. For data gathering, two important BBs have identified and presented to extract information from version control repositories namely VCS Miner and VC Migrator (See Table 2). More specifically, VCS Miner is a more general term that explicitly represents GIT Miner, SVN Miner, and CVS Miner.

Building Blocks for Pre-Processing: As with any dataset, there is certainly a great deal of cleaning and pre-processing required before any real analysis can perform. The pre-processing stage can often take the majority of the time spent on a data analysis project. Having a proper understanding of the required pre-processing steps allows a researcher to speed up the data preparation process as well as to reduce the complexity of the mining process. In this work, seven main BBs have identified for pre-processing: Snapshot Generator, Test Files Remover, Program Model Generator, Dependency Graph Generator, AST Generator, Metrics Generator and Token Generator (see Table 3).

Building Blocks for Clone Detection: Several code clone detection approaches and tools have proposed in the literature spanning from textual to semantic. Designing a clone detection experiments requires identification of a suitable clone detection technique based on the available data set and the output of the pre-processing step. In this work, five central BBs have identified for clone detection: String Based Clone Detector, AST Based Clone Detector, Metrics Based Clone Detector, Graph-Based Clone Detector and Token-Based Clone Detector (see Table 4).

Building Blocks for Clone Evolution: Conducting a comprehensive analysis of clone evolution can uncover the patterns and characteristics exhibited by clones as they evolve within a system. Software practitioners can use the results of this study to understand and to manage the clones more efficiently. In this work, three BBs have identified for post-processing: Genealogy Generator 1, Genealogy Generator 2 and Genealogy Reconstructor (see Table 5).

5 CONCEPTUAL FRAMEWORK OF THE EXPERIMENTAL TESTBED

This section presents the conceptual foundation of a domain-specific framework to support CCDE experiments. The framework adheres to an extensible multi-layered abstraction mechanism that consists of the collection of BBs identified previously. The BBs are systematically organized on top of a collection of basic operators derived from relational algebra.

5.1 Stack of Building Blocks

As depicted in Figure 4, the Building Blocks stack consists of several layers, that are arranged based on the BBs identified in the previous section along with a newly introduced collection of Operators.

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Source code management systems (SCM) characteristically offer a rich version history of software projects. They give crude information about the origins of the code and it's change history. Such information indirectly provide code cloning information at various stages. Therefore, there is a need to gather data from version control repositories such as SVN, CVS and GIT.	**VCS Miner** It mines the version control repository of software systems.	SVN Miner, CVS Miner, GIT Miner	None	Token generator, Snapshot generator, Program model generator, VC migrator, Metrics generator, AST generator	**Xie et al [86]** : This paper analyzed change history of three software systems by mining SVN repositories. **Saha et al. [72]**: This paper analyzed six open source software systems by mining SVN repositories. **Aversano et al. [5]**: This paper extracted code from CVS repositories of two software systems (i.e., ArgoUML and DNSJava) to conduct and empirical study.
Mostly the original software system are stored in traditional SCMs (e.g., SVN, CVS). Decentralized source code management systems (e.g., GIT) can provide richer content histories than the traditional SCMs. Therefore, clone detection algorithms and tools might perform better with a DSCM repository.	**VC Migrator** It migrates data from traditional SCM repository to DCSM repository (e.g., from SVN to GIT)	None	SVN Miner, CVS Miner	Token generator, Snapshot generator, Program model generator, VC migrator, Metrics generator, AST generator	**Rahman et al. [65, 66]**: They migrated SVN and CVS repositories to GIT in order to speed up the process. This paper analyses relationship between cloning and defect proneness.

Table 2: Building blocks for data gathering

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Software evolution experiments need to capture revisions and releases of software systems for a considered time period in order to conduct various code clone detection experiments.	**Snapshot Generator** It can generate a set of snapshots of the software system for a mentioned time period.	release level snapshot generator, revision level snapshot generator	VCS miner	Clone detector, Test files remover	**Xie et al.** [**86**]: This paper extracts snapshots at each revision. **Saha et al.** [**72**]: This paper captures all minor and major releases. **Rahman et al.** [**65, 66**]: This paper requires snapshots in monthly revisions.
Software contains many test files that are used during the development of the system to test the different functionalities. Since test files are frequently copied and modified to test a different case, they can contain many clones.	**Test Files Remover** It removes test files from the subject systems.	None	Snapshot generator	Clone detector	**Barbour et al.** [**9**]: This paper removes test files from ARGOUML and ANT systems. **Xie et al.** [**86**]: This paper removes test files to avoid the clones that are not involved in normal executions.
Logical clones can reveal many business and programming rules that are often not properly documented during software development.Therefore, there is a need for making these rules explicit in order to improve the efficiency in software maintenance.	**Program Model Generator** It extracts the program model from the source code, which consists of methods, entity classes, etc.	None	VCS miner	Text based clone detector	**Qian et al.** [**64**]: This paper extracts a program model from the source code. That is used to analyze the business and programming rules of software applications.
The nodes of a PDG represent the statements and conditions of a program, while edges represent control and data dependencies. Extracting such information is vital to identify similar code fragments.	**Dependency Graph Generator** It generates the PDG of a given source file.	None	VCS miner	Dependency graph based clone detector	**Krinker** [**48**]: Authors have presented an approach based on PDGs, by symbolizing the basic structure of a program together with the corresponding data flow. **Horwitz** [**30**]: This paper investigates both syntactic and semantic differences of various versions of software by exploiting PDGs.

Continued on next page...

Table 3 – Continued

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
In literature, several Abstract Syntax Tree based approaches have been proposed to automate the identification of software clones. During a parsing step, the algorithm should create an AST based representation of the source code.	**AST Generator** It parses the source code and generate AST representation of the source code.	None	VCS miner	AST based clone detector	**Corazza et al. [14]**: This paper proposes a methodology, which investigate ASTs as well as lexical information for discovering software clones up to Type 3. **Tairas and Gray [79]**: This paper proposes a clone detection methodology by utilizing AST representation.
Metrics-based approaches calculate a number of metrics and then compare them rather than directly comparing the source code or ASTs.	**Metrics Generator** It generates code metrics from the source code.	None	VCS miner	Metric-based clone detector	**Anatoniol et al. [4]**: This paper analyzes nineteen releases of Linux kernel to identify code duplication by means of the metrics. **Mayrand et al. [55]**: This paper proposes a metrics-based technique to automatically discover duplicate (or near duplicate) functions in software systems.

Continued on next page...

Table 3 – Continued

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Token-based clone detectors are considered better than simple keyword matches [67]. In these techniques, lexical analysis is primarily used to extract the tokens from the source code.	**Token Generator** It generates tokens from the source code.	None	VCS miner	Token-based clone detector	**Baker [7]**: In research uses a lexical analyzer to divide the lines of source files into tokens. These tokens are then split into parameter tokens and non-parameter tokens. **Li et al. [52]**: In this approach, statements are mapped to numbers by first tokenizing its components, such as variables, operators, constants, functions, keywords, etc. **Wang et al. [85]**: Proposed CCAligner: a token based large-gap clone detector.

Table 3: Building blocks for pre-processing

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Identifying software clones and understanding how software changes between releases are two important issues for maintainers where a text-based approach is likely to be useful.	**String Based Clone Detector** It detects code clones by string matching.	None	Snapshot generator, Program model generator	Clone genealogy generator 1, Clone genealogy generator 2	**Johnson [37]**: This paper considers the source as text and analyze it the way documents are analyzed to detect the code clones. **Ossher et al. [62]**: This paper proposes a file-level clone detection method by combining three simple string matching techniques.
After creating the AST from the source code, similar subtrees need to be identified as code clones. In this case, differences of various coding styles as well as variable names are ignored.	**AST Based Clone Detector** It detects code clones from AST representations.	None	AST generator	Clone genealogy generator 1, Clone genealogy generator 2	**Corazza et al. [14]**: This paper exploits together AST and lexical information to identify software clones. **Baxter et al. [11]**: This paper proposes an AST based methodology to detect near-miss clones.
Metrics-based techniques calculate a number of metrics for code fragments and then compare metrics vectors rather than the source code or ASTs directly.	**Metrics Based Clone Detector** It detects code clones from source code metrics.	None	Metrics generator	Clone genealogy generator 1, Clone genealogy generator 2	**Mayrand et al. [55]**: This paper considers both control flow metrics and data flow metrics to detect code clones.
Semantic approaches on code clone detection vastly rely on the the PDG of the source code. Then subgraph matching techniques are used to identify the code clones in a program.	**Graph Based Clone Detector** It detects code clones from tokens.	None	Program dependency graph generator	Clone genealogy generator 1, Clone genealogy generator 2	**Komondoor and Horwitz [44]**: This paper represents each function in the source code using its PDG, which could then be used to find the code clones. **Krinke [48]**: This paper attempts to identify similar subgraph structures that are stemming from duplicated code.

Continued on next page...

Table 4 – Continued

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Token-based approaches are naturally language-independent and also considered as low-cost. Further it produces faster results because they only need to transform the source code into tokens, without the need to construct ASTs or PDGs.	**Token Based Clone Detector** It detects code clones from tokens.	None	Token generator	Clone genealogy generator 1, Clone genealogy generator 2	**Kamia et al. [40]**: CCFinder is an outcome of this research. **Basit et al. [10]**: This paper proposes a simple and customizable tokenization mechanism for code clone detection. **Jalbert and Bradbury [35]**: This paper proposes a methodology to identify bugs using ConQAT [39], which is a token based clone detector.

Table 4: Building blocks for clone detection

Problem	BB Name and Solution Overview	Alternatives	Predecessors	Follow-ups	References
Code clone genealogies show how clone groups evolve with the evolution of a software system over multiple versions of the program. Therefore, it could provide important insights on the maintenance implications of code clones.	**Genealogy Generator 1** It creates clone genealogies from the detected clones.	Clone genealogy generator 2	Clone detector	None	**Kim and Notkin [42]**: Authors of this paper have presented an approach to discover clone genealogy by exploiting the cloning relationships among all consecutive versions. **Adar and Kim [1]**: They create clone genealogies, allowing visually and programmatically analyzing code clones.
Code clone genealogies show how clone groups evolve with the evolution of a software system over multiple versions of the program. Therefore, it could provide important insights on the maintenance implications of code clones.	**Genealogy Generator 2** It removes invalid clones prior to generating clone genealogies.	Clone genealogy generator 1	Clone detector	Genealogy reconstructor	**Xie et al. [86]**: Prior to building the clone genealogies, this study removes all the unchanged and invalid (i.e., code segments belong to the same method).
Clone genealogy generation is usually done using tools such as gCad. Such tools could sometimes provide erroneous results. Therefore, manual verification is indeed important.	**Genealogy Reconstructor** It manually verifies the correctness of the results and remove the false positives.	None	Clone genealogy generator 1/2	None	**Saha et al. [72]**: They run gCad for the second time to reconstruct the genealogies after removing the false positives manually.

Table 5: Building blocks for clone evolution

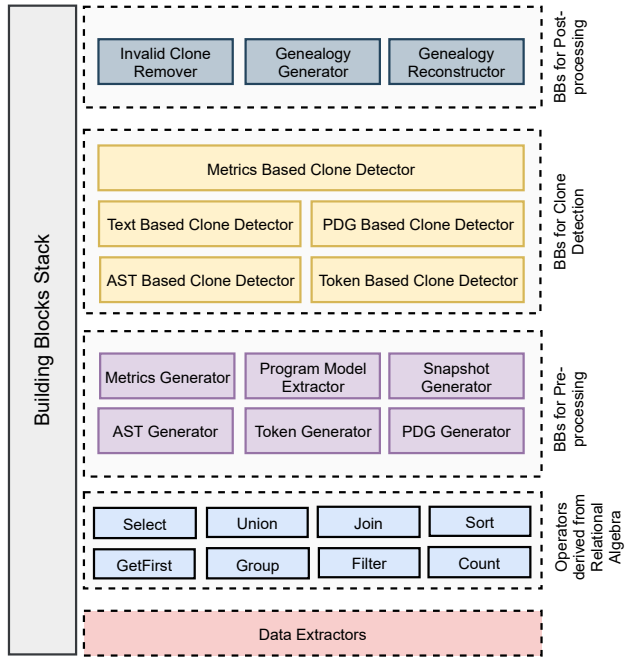

Figure 4. Building blocks stack supported by the operators derived from relational algebra

These operators are directly derived from relational algebra. Relational algebra is a procedural query language, which operates on input relations. It consists with a set of fundamental operators such as `select`, `project`, `union` and `cartesian product`. Though several relational algebra theorems do not strictly hold in query languages such as SQL and LINQ, they are the native implementations of the underline concept of relational algebra. Therefore, we borrowed some ideas from such languages to identify basic operators supported by relational algebra. In this paper, operators such as `filter`, `select`, `join`, `sort`, `count`, etc. has been categorized as basic level operators. Such operators are useful in conducting CCDE experiments.

5.2 Architectural Overview of the Experimental Testbed

Figure 5 presents the architectural overview of the experimental testbed to conduct CCDE experiments. It consists of two main components: BBs repository, and workflow composition and execution engine. BBs repository contains all the BBs (i.e., BBs for data extraction, pre-processing, clone detection, and clone evolution) and a useful collection of *Operators* that are described previously. The purpose of

Operators is to facilitate the basic functionalities such as counting or filtering. Once the BBs and Operators are defined, CCDE experiments can be accomplished by pipelining the required BBs and *operators*. Workflow composition and execution engine is responsible for translating the CCDE workflow defined by a user into an executable process. Finally, the analysis results will be presented to the user.

For the workflow generation, all BBs and Operators are defined directly on an underlying logical representation, a static grammar. Static grammar consists of the production rules to combine BBs and Operators into a meaningful workflow, which strictly follows the connections in Figure 3. For example, *AST based clone detector* can be directly composed with the *AST generator*. However, it cannot be composed with *PDG generator*. Static grammar has to be defined manually and should be evolved with the introduction of new BBs and Operators.

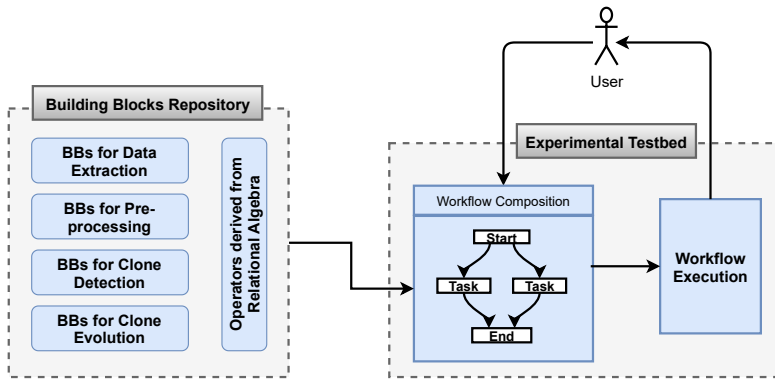

Figure 5. Architectural overview of the experimental testbed

6 VALIDATION

Our vision is to introduce a collection of reusable BBs that are derived from the state-of-the-art code clone detection and evolution research and efficiently utilize them in developing an experimental testbed to conduct CCDE experiments systematically and conveniently. In this section, we sought to validate the two research questions. RQ1 mainly focuses on identifying building blocks from existing CCDE experiments, which could reuse in new ventures. For that, a case study based evaluation is employed to show how a particular CCDE experiment can represent by utilizing the identified BBs. To validate RQ2, a simple prototype was implemented to demonstrate how to develop an experimental testbed to utilize BBs effectively. The prototype was validated with a usage scenarios for three open source projects. Finally, the extensibility of the proposed approach in conducting a diverse range of software analytics experiments is examined.

6.1 Reusability of BBs in CCDE Experiments

As described previously, the BBs have identified by the literature survey conducted on the papers published in ICSE, ICSM, MSR and FSE conferences for the last eight years. Therefore, for the validation purpose, four journal papers on code clone detection have selected as case studies. Then, each experimental procedures were represented as a workflow by utilizing the identified BBs. For the selected case studies, it was evident that one experiment can be fully expressed and three experiments can be partially represented using BBs.

Case Study 1 – Kontogiannis et al. [45]

Summary: Authors of this paper have presented a number of pattern matching techniques by using ASTs as the code representation scheme that could use for both code-to-code as well as concept-to-code matching. Metric-based clone detection technique has used in the study by taking three medium-sized C programs (i.e., tcsh, bash and CLIPs) as the subject systems. First, the source code is parsed to create the Abstract Syntax Tree (AST). Five different metrics have calculated for every statement, block, function, and file stored as annotations in the nodes of the AST. As the next step, a reference table was maintained, which consists of source code entities sorted by their associated metric values.

Representation Using BBs: In this experiment, VCS miner considered the BB for data gathering. Pre-processing step is covered with two BBs namely AST generator and Metrics generator. Metrics based clone detector is the responsible BB in the clone detection phase. Thus, the experimental design of the above paper can be fully represented using four main BBs, as shown in Figure 6.

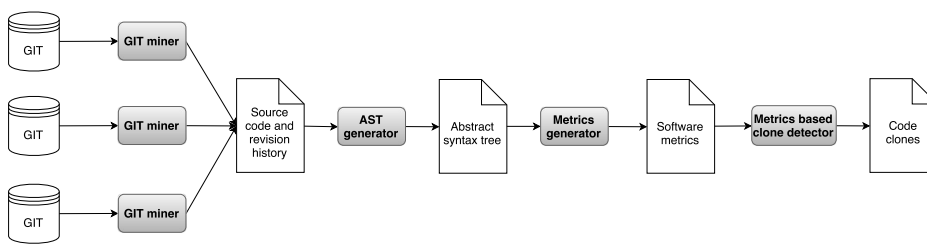

Figure 6. Kontogiannis's [45] approach using BBs

Case Study 2 – Anatoniol et al. [4]

Summary: This paper studies the evolution of code duplications in the Linux kernel. The paper followed a functional level metric-based approach to analyze nineteen releases to identify code duplication among Linux subsystems.

Representation Using BBs: Figure 7 is an example how BBs can be used to partially representing a previously conducted experimental design. Authors of this paper have described mechanisms to handle preprocessor directives as well as to handle the functions in the C code of the Linux kernel. Such tasks come under the above mentioned Pre-processing activity. However, at the moment, the exact BB to perform this task is not available in our BBs catalog. As stated before, our approach will evolve with time and build its BBs catalog. Therefore, one can define a new BB and add to our BBs catalog. However, the above experiment can partially represent by utilizing VCS miner, Metrics generator and Metrics based clone detector.

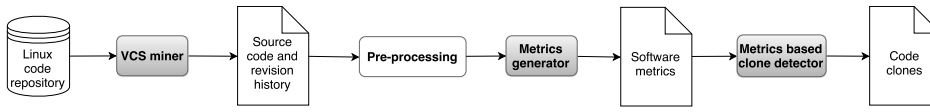

Figure 7. Anatoniol's [4] approach using BBs

Case Study 3 – Geiger et al. [25]

Summary: In this paper, the authors examined whether a correlation exists between code clones and code change. The steps of this research include code clone detection, categorization into clone types, extraction of change couplings, and computing a relation metric. The proposed framework has validated with the Mozilla project. The results show that a reasonable number of cases can found where such a relation exists.

Representation Using BBs: Part of the experiment of this research can represent using BBs, as shown in Figure 8. In this paper, authors have used CCFinder as the clone detection tool. In Figure 8, we further describe the tasks in CCFinder as a BPMN 2.0 subprocess. VCS miner, Token generator, and Token-based clone detector have used in this expanded representation.

Case Study 4 – Kanwal et al. [41]

Summary: This paper investigates the evolution of structural clones by conducting a longitudinal analysis of several versions of Java systems. The authors have defined structural clones and their evolution patterns in a formal notation. The trends in the patterns reveal that evolutionary characteristics of structural clones can facilitate better clone management systems.

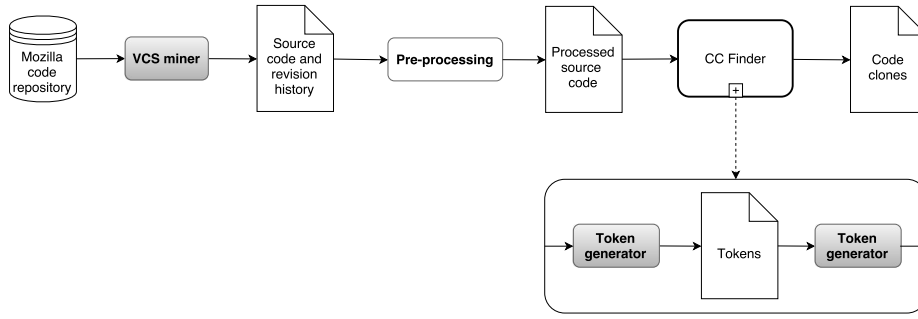

Figure 8. Geiger's [25] approach using BBs

Representation Using BBs: As depicted in Figure 9, the experiment can be effectively represent with the identified BBs. The above experiment can be partially represented using VCS Miner, Token generator, Token based clone detector, and Genealogy generator.

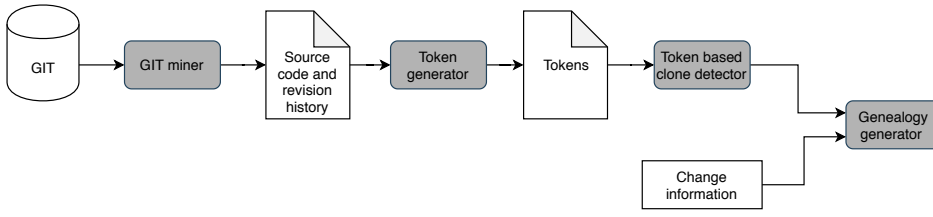

Figure 9. Kanwal's [41] approach using BBs

Summary of the evaluation results is shown in Table 6. Based on the case studies, the RQ1 can be addressed, and we claim that it is a serious first proof of the usefulness of the proposed BBs. The selection of case studies is based on the clone detection and evolution experiments spanning from the year 1996 to 2019 denoting the applicability of the proposed approach in the future clone detection and evolution experiments.

6.2 Usage Scenario in the Experimental Testbed

Clone analysis over multiple versions and releases is a major component in many CCDE experiments [75, 56]. Such studies would reveal the trends over time as well as the relationship between code size and the number of code clones for large-scale software projects [13]. Below we show how to use the experimental testbed to find the code clone percentage over multiple versions of a software project.

In order to find code clones over multiple versions, the following tasks have to perform in the given order. First, project history for a given version/release needs

Title of the journal paper	**Used BBs**	**Graphical Representation**
Pattern matching for clone and concept detection. Kontogiannis et al. [45]	VCS Miner AST Generator Metrics Generator Metrics Based Clone Detector	Figure 4
Analyzing cloning evolution in the Linux kernel. Anatoniol et al. [4]	VCS Miner Metrics Generator Metrics based Clone Detector	Figure 5
Relation of code clones and change couplings Geiger et al. [25]	VCS Miner Token Generator Token Based Clone Detector	Figure 6
Evolutionary Perspective of Structural Clones in Software Kanwal et al. [41]	VCS Miner Token Generator Token Based Clone Detector Genealogy generator	Figure 7

Table 6. Summary of the case study based validation

to be extracted from the version control repository using VCS Miner. Second, it is converted to an intermediate data-model using one of the pre-processing BBs. Then the results are fed to a Clone Detector to detect the duplicates. Finally, the steps mentioned above are repeated for several versions of the software system. In the prototype implementation, the BBs can be pipelined as a workflow and run the analysis. Additional BBs (i.e., filter, loop) can be implemented to facilitate rich analyses based on complex conditions. As such, a user needs to drag the BBs to the canvas and combine them using linkers and run it. Three Apache projects have been selected for the experiment; Apache Commons Lang[5], Apache Tomcat[6] and Apache Wink[7]. Figure 10 presents the cloning behavior for the years 2014–2016.

However, by no means, this is a complex CCDE experiment. But, still, it answers RQ2 by evidently demonstrating how BBs can be used in the proposed experimental testbed to produce useful insights to the researchers.

6.3 Extensibility of Experiment Testbed for Software Engineering Experiments

The core idea behind BBs and the conceptual framework of the experimental testbed is not strictly limited to CCDE research. The proposed architecture of the testbed along with the composition logic of BBs provide versatility for extending the experimental testbed for other types of software engineering experiments. However,

[5] https://commons.apache.org/proper/commons-lang/
[6] http://tomcat.apache.org/
[7] https://wink.apache.org/

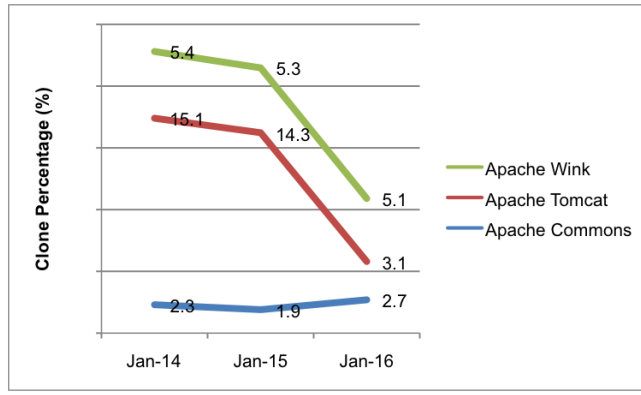

Figure 10. Clone percentage for 2014–2016

it requires a formal arrangement of BBs into several layers. Figure 11 presents the proposed extended stack of BBs that could be used in different software engineering experiments. The extended BBs stack for software analytics has multiple layers: Primary BBs, Secondary BBs, and Advanced BBs.

Below we demonstrate how to build the logic to perform a software analysis task by utilizing the BBs from the BBs Stack.

Analysis Task: Finding critical issues resolved by most frequent committer in a project.

Background: Measuring the performance of the developers who work in a project is a challenging task for the project managers when the team size is large and the nature of the project is complex. However, it is notable that total lines of codes, the number of bugs fixed, the total number of commits, or a combination of them could produce useful insights into performance.

Implementation Using BBs: Figure 12 presents how to utilize the BBs to perform the above analysis task. It demonstrates how the data is integrated from both version control and bug tracking repositories to find how many critical bugs have been fixed by the most frequent committer.

We further tested the above scenario with three open source projects by using the prototype implementation. The prototype allows users to utilize the BBs to perform the tasks directly. Therefore, it provides a great level of convenience to the users. Summary of the experimental results present in Table 7.

As shown in the Figure 12, FindMax, which is a Secondary BB, is formulated by utilizing three Primary BBs. Thus, Secondary BBs, on the ohter hand, can be considered as composite BBs.

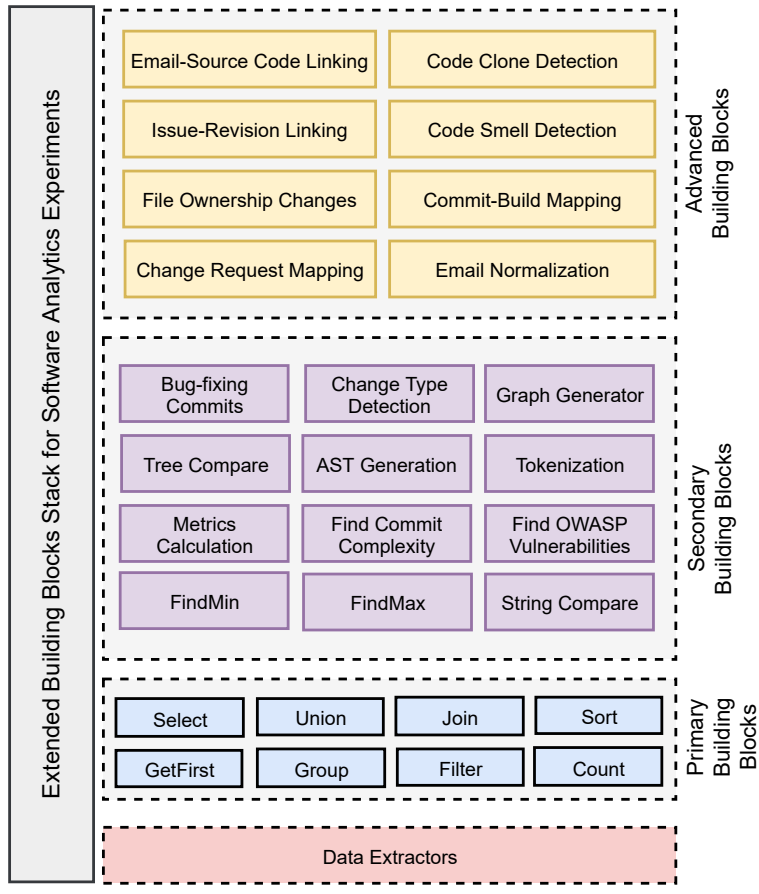

Figure 11. Extended building blocks stack for software analytics

7 DISCUSSION

In this paper, we provide a catalog of Building Blocks on which the clone detection research can be carried out. A particular BB represents a specific analysis task in any CCDE experiment. Based on that, we demonstrated that such distinctive BBs could properly arrange as workflows to perform a wide range of CCDE experiments. From the selected case studies for the validation, it was evident that CCDE experiments can either wholly or partially represent by means of BBs. Therefore, this approach is a step towards standardization of CCDE research by providing a structured way to conduct experiments. Our approach has multifold benefits and is worth further exploration.

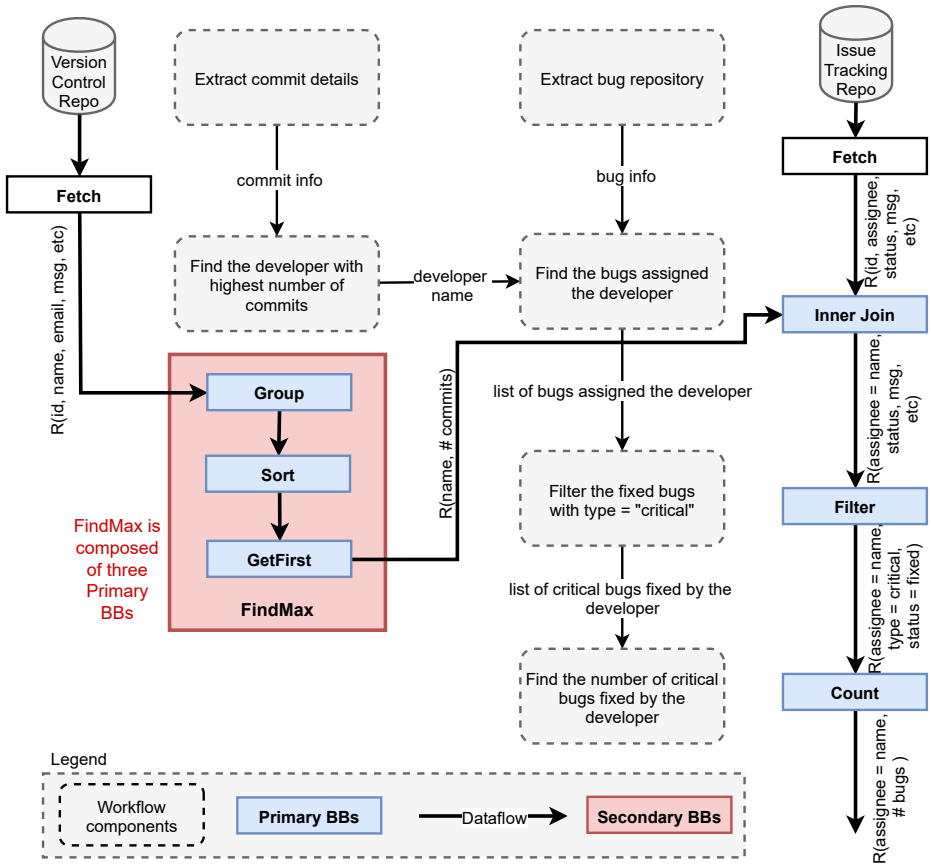

Figure 12. Finding the number of critical issues resolved by the most frequent committer in a project

Guideline for Novice Researchers. This paper does not target providing a comprehensive literature review in the area of CCDE. Most importantly, it presents some useful conceptual and practical insights to novice researchers by allowing them to use BBs as a guide to carrying out CCDE experiments. Novice researchers can make use of BBs to conduct experiments in a quick and community accepted way. Having a prior understanding of the BBs will help them comprehend the published CCDE research approaches and recognize the essential background requirements; hence, can better plan their experiments. Further analyzing the usages of those BBs in different analysis scenarios will help them in running successful experiments.

Helping Overcome Common Problems in CCDE Experiments. Conducting CCDE experiments presents a number of common difficulties and challenges to

Apache Project	No. of Commits	Frequent Developer	No. of Bug Fixes
Gora	1 053	Developer A	52
Commons-lang	5 171	Developer B	4
IO	2 091	Developer C	0
Winx	1 312	Developer D	2

Table 7. Summary of the experimental results

researchers such as:

1. mechanism to locate the repositories to gather accurate and timely data,
2. filtering or converting such data to different formats,
3. exploring various analysis to be performed on such data, and
4. effectively running such analyses.

The concept of Activities and Building Blocks is beneficial to overcome such exertions. For example, BBs for data gathering facilitates a mechanism to locate and extract data from repositories. Similarly, BBs for pre-processing provide ways to convert and filter data. BBs for clone detection solves the difficulty in exploring distinctive analysis on such data. In that way, our approach simplifies the challenges mentioned above and provides a structured way to conduct software analysis experiments.

Facilitating Comparison. Several imperative systematic literature reviews have published on software clones in general and software clone detection in particular. These approaches typically focus on only some traits of categorization, and most of them do not rely on an explicit high level meta-model. Therefore, there is a need of a model, which facilitates the comparison of different clone detection approaches at the experimental level. In this paper, we present a meta-model infrastructure for representing, combining and comparing such experiments in a structured way.

Fostering the Replication of Studies. The replication of such studies is just as fundamental and is one of the main threats to validity that empirical software engineering suffers. Such threats are manifold and range from lack of independent validation of the results, unavailability of the tools and methodologies used, to no impossibility to generalize the gained knowledge. Though this paper does not provide a fully functional framework for replication, still it presents ways to better plan the replication studies and reveal the imprecise descriptions in *Methodology* sections of research publications.

Based on the nature of the BBs, it is important to realize that the proposed approach works only with syntactically similar code clones.For example, as described in the BBs for pre-processing, the entire detection process is facilitated by ASTs, PDGs, tokens, metrics, program models, and snapshots. Thus the BBs for clone detection facilitates only the syntactically similar code clones. However, the detec-

tion of semantically similar code clones requires a new set of BBs that are capable of inferring the associations across functionally similar code clones.

Several recent studies have reported on cross-language code clone detection [58, 88, 59]. For example, LICCA, a tool for cross-language clone detection [82] is based on a tree-based intermediate representation of the source code. Thus, the proposed BBs for pre-processing can be used for this purpose. However, this direction has to further investigate to identify a useful set of BBs for cross-language clone detection.

Besides, visualization of the results produced by software analytics is considered important nowadays [81, 19]. Also, recent studies have highlighted the importance of visualizing the differences between the versions of software models [61]. Thus, the proposed BBs stack for software analytics has provisions to augment with new BBs that could be used to facilitate the visualization aspects of software analytics experiments.

8 CONCLUSIONS

This paper introduced a concrete set of formal constructs, which we refer to as *Building Blocks*, which can be used to conduct various CCDE experiments. These *Building Blocks* provide a structured way to conduct experiments, hence it offers direct solutions to everyday challenges in code clone detection, such as accurate data collection, data cleaning, and selecting proper CCD algorithms. Our goal is not to introduce novel CCD algorithms or report the loopholes in the existing CCDE research, but to provide a systematic understanding of how CCDE experiments are conducted in practice by utilizing the identified *Building Blocks*. *Building Blocks* are represented using both textual and graphical representation, which provide means to software researchers to conduct or replicate CCDE experiments in an unambiguous manner. The conceptual framework of the experimental testbed indicates the usefulness and the replication capabilities of *Building Blocks* and is proven useful in conducting CCDE experiments. Besides, this paper presents how the stack of *Building Blocks* can be extended to facilitate a wide range of software analytics experiments beyond CCDE.

Future work of this research will focus on enhancing the experimental testbed to a point where we can conduct a field study with professional software practitioners in the industry. By doing that it is expected to obtain the future potentials and the limitations of the experimental testbed in practice. In that way, useful insights can be gained to convert our testbed to a full-fledged software evolution analysis testbed.

Acknowledgements

The authors of this paper gratefully acknowledge the financial support provided by the National Research Council of Sri Lanka (Grant No. NRC 15-74). We thankfully

acknowledge the insights and expertise provided by the colleagues at SEAL Lab at the University of Zurich.

REFERENCES

[1] Adar, E.—Kim, M.: SoftGUESS: Visualization and Exploration of Code Clones in Context. 29th International Conference on Software Engineering (ICSE '07), 2007, pp. 762–766, doi: 10.1109/ICSE.2007.76.

[2] Agrawal, A.—Yadav, S. K.: A Hybrid-Token and Textual Based Approach to Find Similar Code Segments. 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), 2013, pp. 1–4, doi: 10.1109/ICCCNT.2013.6726700.

[3] Ain, Q. U.—Butt, W. H.—Anwar, M. W.—Azam, F.—Maqbool, B.: A Systematic Review on Code Clone Detection. IEEE Access, 2019, Vol. 7, pp. 86121–86144, doi: 10.1109/ACCESS.2019.2918202.

[4] Antoniol, G.—Villano, U.—Merlo, E.—Di Penta, M.: Analyzing Cloning Evolution in the Linux Kernel. Information and Software Technology. Vol. 44, 2002, No. 13, pp. 755–765, doi: 10.1016/S0950-5849(02)00123-4.

[5] Aversano, L.—Cerulo, L.—Di Penta, M.: How Clones Are Maintained: An Empirical Study. 11th European Conference on Software Maintenance and Reengineering (CSMR '07), 2007, pp. 81–90, doi: 10.1109/CSMR.2007.26.

[6] Baker, B. S.: A Program for Identifying Duplicated Code. Computing Science and Statistics, 1993, pp. 49–49.

[7] Baker, B. S.: On Finding Duplication and Near-Duplication in Large Software Systems. Proceedings of 2nd Working Conference on Reverse Engineering, 1995, pp. 86–95, doi: 10.1109/WCRE.1995.514697.

[8] Barbour, L.—Khomh, F.—Zou, Y.: An Empirical Study of Faults in Late Propagation Clone Genealogies. Journal of Software: Evolution and Process, Vol. 25, 2013, No. 11, pp. 1139–1165, doi: 10.1002/smr.1597.

[9] Barbour, L.—Khomh, F.—Zou, Y.: Late Propagation in Software Clones. 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 273–282, doi: 10.1109/ICSM.2011.6080794.

[10] Basit, H. A.—Jarzabek, S.: Efficient Token Based Clone Detection with Flexible Tokenization. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), 2007, pp. 513–516, doi: 10.1145/1287624.1287698.

[11] Baxter, I. D.—Yahin, A.—Moura, L.—Sant'Anna, M.—Bier, L.: Clone Detection Using Abstract Syntax Trees. International Conference on Software Maintenance, 1998, pp. 368–377, doi: 10.1109/ICSM.1998.738528.

[12] Buehrer, G.—Weide, B. W.—Sivilotti, P. A. G.: Using Parse Tree Validation to Prevent SQL Injection Attacks. 5th International Workshop on Software Engineering and Middleware (SEM '05), 2005, pp. 106–113, doi: 10.1145/1108473.1108496.

[13] Chen, X.—Wang, A. Y.—Tempero, E.: A Replication and Reproduction of Code Clone Detection Studies. Proceedings of the Thirty-Seventh Australasian Computer Science Conference (ACSC 2014), Conferences in Research and Practice in Information Technology (CRPIT), Vol. 147, 2014, pp. 105–114.

[14] Corazza, A.—Di Martino, S.—Maggio, V.—Scanniello, G.: A Tree Kernel Based Approach for Clone Detection. IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–5, doi: 10.1109/ICSM.2010.5609715.

[15] Cordy, J. R.: Comprehending Reality – Practical Barriers to Industrial Adoption of Software Maintenance Automation. 11th IEEE International Workshop on Program Comprehension, 2003, pp. 196–205, doi: 10.1109/WPC.2003.1199203.

[16] Cosentino, V.—Izquierdo, J. L. C.—Cabot, J.: Gitana: A SQL-Based Git Repository Inspector. In: Johannesson, P., Lee, M., Liddle, S., Opdahl, A., Pastor López, Ó. (Eds.): Conceptual Modeling (ER 2015). Springer, Cham, Lecture Notes in Computer Science, Vol. 9381, 2015, pp. 329–343, doi: 10.1007/978-3-319-25264-3_24.

[17] D'Ambros, M.: Supporting Software Evolution Analysis with Historical Dependencies and Defect Information. IEEE International Conference on Software Maintenance, 2008, pp. 412–415, doi: 10.1109/ICSM.2008.4658092.

[18] Deelman, E.—Gil, Y.: Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges. 2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science '06), 2006, p. 144, doi: 10.1109/E-SCIENCE.2006.261077.

[19] Dominic, J.—Tubre, B.—Houser, J.—Ritter, C.—Kunkel, D.—Rodeghero, P.: Program Comprehension in Virtual Reality. Proceedings of the 28th International Conference on Program Comprehension (ICPC '20), 2020, pp. 391–395, doi: 10.1145/3387904.3389287.

[20] Dou, W.—Cheung, S. C.—Gao, C.—Xu, C.—Xu, L.—Wei, J.: Detecting Table Clones and Smells in Spreadsheets. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016), 2016, pp. 787–798, doi: 10.1145/2950290.2950359.

[21] Ducasse, S.—Rieger, M.—Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. IEEE International Conference on Software Maintenance (ICSM '99), 1999, pp. 109–118, doi: 10.1109/ICSM.1999.792593.

[22] Dyer, R.—Nguyen, H. A.—Rajan, H.—Nguyen, T. N.: Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 422–431, doi: 10.1109/ICSE.2013.6606588.

[23] Ferrante, J.—Ottenstein, K. J.—Warren, J. D.: The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 9, 1987, No. 3, pp. 319–349, doi: 10.1145/24039.24041.

[24] Fowler, M.: Refactoring: Improving the Design of Existing Code. Pearson Education India, 1999.

[25] Geiger, R.—Fluri, B.—Gall, H. C.—Pinzger, M.: Relation of Code Clones and Change Couplings. In: Baresi, L., Heckel, R. (Eds.): Fundamental Approaches to Software Engineering (FASE 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3922, 2006, pp. 411–425, doi: 10.1007/11693017_31.

[26] Göde, N.—Harder, J.: Clone Stability. 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011, pp. 65–74, doi: 10.1109/CSMR.2011.11.

[27] Gousios, G.—Spinellis, D.: Conducting Quantitative Software Engineering Studies with Alitheia Core. Empirical Software Engineering, Vol. 19, 2014, No. 4, pp. 885–925, doi: 10.1007/s10664-013-9242-3.

[28] Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997, doi: 10.1017/CBO9780511574931.

[29] Higo, Y.—Kusumoto, S.: Code Clone Detection on Specialized PDGs with Heuristics. 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011, pp. 75–84, doi: 10.1109/CSMR.2011.12.

[30] Horwitz, S.: Identifying the Semantic and Textual Differences Between Two Versions of a Program. ACM SIGPLAN Notices, Vol. 25, 1990, No. 6, pp. 234–245, doi: 10.1145/93542.93574.

[31] Hu, Y.—Wang, H.—Zhang, Y.—Li, B.—Gu, D.: A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. IEEE Transactions on Software Engineering, Vol. 47, 2021, No. 6, pp. 1241–1258, doi: 10.1109/TSE.2019.2918326.

[32] Hu, Y.—Zhang, Y.—Li, J.—Wang, H.—Li, B.—Gu, D.: BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 104–114, doi: 10.1109/ICSME.2018.00019.

[33] Hummel, B.—Juergens, E.—Heinemann, L.—Conradt, M.: Index-Based Code Clone Detection: Incremental, Distributed, Scalable. 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–9, doi: 10.1109/ICSM.2010.5609665.

[34] Islam, J. F.—Mondal, M.—Roy, C. K.—Schneider, K. A.: Comparing Bug Replication in Regular and Micro Code Clones. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 81–92, doi: 10.1109/ICPC.2019.00022.

[35] Jalbert, K.—Bradbury, J. S.: Using Clone Detection to Identify Bugs in Concurrent Software. 2010 IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–5, doi: 10.1109/ICSM.2010.5609529.

[36] Johnson, J. H.: Identifying Redundancy in Source Code Using Fingerprints. Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering (CASCON '93), Vol. 1, 1993, pp. 171–183.

[37] Johnson, J. H.: Substring Matching for Clone Detection and Change Tracking. 1994 International Conference on Software Maintenance, 1994, pp. 120–126, doi: 10.1109/ICSM.1994.336783.

[38] Johnson, J.: Visualizing Textual Redundancy in Legacy Source. Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering (CASCON '94), 1994, pp. 32–41.

[39] Juergens, E.—Deissenboeck, F.—Hummel, B.: CloneDetective – A Workbench for Clone Detection Research. Proceedings of the IEEE 31st International Conference on Software Engineering, 2009, pp. 603–606, doi: 10.1109/ICSE.2009.5070566.

[40] 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, Vol. 28, 2002, No. 7, pp. 654–670, doi: 10.1109/TSE.2002.1019480.

[41] Kanwal, J.—Maqbool, O.—Basit, H. A.—Sindhu, M. A.: Evolutionary Perspective of Structural Clones in Software. IEEE Access, Vol. 7, 2019, pp. 58720–58739, doi: 10.1109/ACCESS.2019.2913043.

[42] Kim, M.—Notkin, D.: Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clone. ACM SIGSOFT Software Engineering Notes, Vol. 30, 2005, No. 4, pp. 1–5, doi: 10.1145/1083142.1083146.

[43] Kim, M.—Sazawal, V.—Notkin, D.—Murphy, G.: An Empirical Study of Code Clone Genealogies. ACM SIGSOFT Software Engineering Notes, Vol. 30, 2005, No. 5, pp. 187–196, doi: 10.1145/1095430.1081737.

[44] Komondoor, R.—Horwitz, S.: Using Slicing to Identify Duplication in Source Code. In: Cousot, P. (Ed.): Static Analysis (SAS 2001). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2126, 2001, pp. 40–56, doi: 10.1007/3-540-47764-0_3.

[45] Kontogiannis, K. A.—DeMori, R.—Merlo, E.—Galler, M.—Bernstein, M.: Pattern Matching for Clone and Concept Detection. Automated Software Engineering, Vol. 3, 1996, No. 1-2, pp. 77–108, doi: 10.1007/BF00126960.

[46] Koschke, R.: Survey of Research on Software Clones. In: Koschke, R., Merlo, E., Walenstein, A. (Eds.): Duplication, Redundancy, and Similarity in Software. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl Seminar Proceedings, 2007, pp. 368–377.

[47] Krinke, J.: A Study of Consistent and Inconsistent Changes to Code Clones. 14th Working Conference on Reverse Engineering (WCRE 2007), 2007, pp. 170–178, doi: 10.1109/WCRE.2007.7.

[48] Krinke, J.: Identifying Similar Code with Program Dependence Graphs. Eighth Working Conference on Reverse Engineering, 2001, pp. 301–309, doi: 10.1109/WCRE.2001.957835.

[49] Lee, S.—Jeong, I.: SDD: High Performance Code Clone Detection System for Large Scale Source Code. Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05), 2005, pp. 140–141, doi: 10.1145/1094855.1094903.

[50] Leitão, A.: Detection of Redundant Code Using R^2D^2. Software Quality Journal, Vol. 12, 2004, No. 4, pp. 361–382, doi: 10.1023/B:SQJO.0000039793.31052.72.

[51] Li, L.—Feng, H.—Zhuang, W.—Meng, N.—Ryder, B.: CCLearner: A Deep Learning-Based Clone Detection Approach. 2017 IEEE International Conference on

Software Maintenance and Evolution (ICSME), 2017, pp. 249–260, doi: 10.1109/ICSME.2017.46.

[52] 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, Vol. 32, 2006, No. 3, pp. 176–192, doi: 10.1109/TSE.2006.28.

[53] Lu, S.—Zhang, J.: Collaborative Scientific Workflows Supporting Collaborative Science. International Journal of Business Process Integration and Management (IJBPIM), Vol. 5, 2011, No. 2, pp. 185-199, doi: 10.1504/IJBPIM.2011.040209.

[54] Manber, U.—Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing, Vol. 22, 1993, No. 5, pp. 935–948, doi: 10.1137/0222058.

[55] Mayrand, J.—Leblanc, C.—Merlo, E. M.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96), 1996, pp. 244–253, doi: 10.1109/ICSM.1996.565012.

[56] Mui, H. H.—Zaidman, A.—Pinzger, M.: Studying Late Propagations in Code Clone Evolution Using Software Repository Mining. Electronic Communications of the EASST, Vol. 63, 2014, doi: 10.14279/tuj.eceasst.63.916.

[57] Nafi, K. W.—Kar, T. S.—Roy, B.—Roy, C. K.—Schneider, K. A.: CLCDSA: Cross Language Code Clone Detection Using Syntactical Features and API Documentation. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1026–1037, doi: 10.1109/ASE.2019.00099.

[58] Nafi, K. W.—Roy, B.—Roy, C. K.—Schneider, K. A.: A Universal Cross Language Software Similarity Detector for Open Source Software Categorization. Journal of Systems and Software, Vol. 162, 2020, Art. No. 110491, doi: 10.1016/j.jss.2019.110491.

[59] Nichols, L.—Emre, M.—Hardekopf, B.: Structural and Nominal Cross-Language Clone Detection. In: Hähnle, R., van der Aalst, W. (Eds.): Fundamental Approaches to Software Engineering (FASE 2019). Springer, Cham, Lecture Notes in Computer Science, Vol. 11424, 2019, pp. 247–263, doi: 10.1007/978-3-030-16722-6_14.

[60] Noonan, R. E.: An Algorithm for Generating Abstract Syntax Trees. Computer Languages, Vol. 10, 1985, No. 3-4, pp. 225–236, doi: 10.1016/0096-0551(85)90018-9.

[61] Ondik, J.—Rástočný, K.: Interactive Visualization of Differences Between Software Model Versions. Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019), 2019, pp. 264-271, doi: 10.5220/0007345502640271.

[62] Ossher, J.—Sajnani, H.—Lopes, C.: File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly. 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 283–292, doi: 10.1109/ICSM.2011.6080795.

[63] Pate, J. R.—Tairas, R.—Kraft, N. A.: Clone Evolution: A Systematic Review. Journal of Software: Evolution and Process, Vol. 25, 2013, No. 3, pp. 261–283, doi: 10.1002/smr.579.

[64] Qian, W.—Peng, X.—Xing, Z.—Jarzabek, S.—Zhao, W.: Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules. 2013

29th IEEE International Conference on Software Maintenance, 2013, pp. 40–49, doi: 10.1109/ICSM.2013.15.

[65] Rahman, F.—Bird, C.—Devanbu, P.: Clones: What is that Smell? 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010, pp. 72–81, doi: 10.1109/MSR.2010.5463343.

[66] Rahman, F.—Bird, C.—Devanbu, P.: Clones: What is that Smell? Empirical Software Engineering, Vol. 17, 2012, No. 4-5, pp. 503–530, doi: 10.1007/s10664-011-9195-3.

[67] Rattan, D.—Bhatia, R.—Singh, M.: Software Clone Detection: A Systematic Review. Information and Software Technology, Vol. 55, 2013, No. 7, pp. 1165–1199, doi: 10.1016/j.infsof.2013.01.008.

[68] Roy, C. K.—Cordy, J. R.: A Survey on Software Clone Detection Research. Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007, pp. 64–68.

[69] Roy, C. K.—Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. 2008 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181, doi: 10.1109/ICPC.2008.41.

[70] Roy, C. K.—Cordy, J. R.—Koschke, R.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming, Vol. 74, 2009, No. 7, pp. 470–495, doi: 10.1016/j.scico.2009.02.007.

[71] Saha, R. K.—Asaduzzaman, M.—Zibran, M. F.—Roy, C. K.—Schneider, K. A.: Evaluating Code Clone Genealogies at Release Level: An Empirical Study. 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), 2010, pp. 87–96, doi: 10.1109/SCAM.2010.32.

[72] Saha, R. K.—Roy, C. K.—Schneider, K. A.—Perry, D. E.: Understanding the Evolution of Type-3 Clones: An Exploratory Study. 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 139–148, doi: 10.1109/MSR.2013.6624021.

[73] Saini, V.—Farmahinifarahani, F.—Lu, Y.—Baldi, P.—Lopes, C. V.: Oreo: Detection of Clones in the Twilight Zone. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), 2018, pp. 354–365, doi: 10.1145/3236024.3236026.

[74] Sajnani, H.—Saini, V.—Svajlenko, J.—Roy, C. K.—Lopes, C. V.: SourcererCC: Scaling Code Clone Detection to Big-Code. Proceedings of the 38th International Conference on Software Engineering (ICSE '16), 2016, pp. 1157–1168, doi: 10.1145/2884781.2884877.

[75] Schwarz, N.—Lungu, M.—Robbes, R.: On How Often Code is Cloned Across Repositories. Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 1289–1292, doi: 10.1109/ICSE.2012.6227097.

[76] Sheneamer, A.—Kalita, J.: A Survey of Software Clone Detection Techniques. International Journal of Computer Applications, Vol. 137, 2016, No. 10, pp. 1–21, doi: 10.5120/IJCA2016908896.

[77] Sokol, F. Z.—Aniche, M. F.—Gerosa, M. A.: MetricMiner: Supporting Researchers in Mining Software Repositories. 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 142–146, doi: 10.1109/SCAM.2013.6648195.

[78] Stevens, R.—De Roover, C.: Querying the History of Software Projects Using QWALKEKO. 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 585–588, doi: 10.1109/ICSME.2014.101.

[79] Tairas, R.—Gray, J.: Phoenix-Based Clone Detection Using Suffix Trees. Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE 44), 2006, pp. 679–684, doi: 10.1145/1185448.1185597.

[80] Taylor, I. J.—Deelman, E.—Gannon, D. B.—Shields, M. (Eds.): Workflows for e-Science: Scientific Workflows for Grids. Springer Publishing Company, Incorporated, 2014, doi: 10.1007/978-1-84628-757-2.

[81] Vincur, J.—Navrat, P.—Polasek, I.: VR City: Software Analysis in Virtual Reality Environment. 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2017, pp. 509–516, doi: 10.1109/QRS-C.2017.88.

[82] Vislavski, T.—Rakić, G.—Cardozo, N.—Budimac, Z.: LICCA: A Tool for Cross-Language Clone Detection. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 512–516, doi: 10.1109/SANER.2018.8330250.

[83] Wahler, V.—Seipel, D.—Wolff, J.—Fischer, G.: Clone Detection in Source Code by Frequent Itemset Techniques. Fourth IEEE International Workshop on Source Code Analysis and Manipulation, 2004, pp. 128–135, doi: 10.1109/SCAM.2004.6.

[84] Walker, A.—Cerny, T.—Song, E.: Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. ACM SIGAPP Applied Computing Review, Vol. 19, 2019, No. 4, pp. 28–39, doi: 10.1145/3381307.3381310.

[85] Wang, P.—Svajlenko, J.—Wu, Y.—Xu, Y.—Roy, C. K.: CCAligner: A Token Based Large-Gap Clone Detector. Proceedings of the 40th International Conference on Software Engineering (ICSE '18), 2018, pp. 1066–1077, doi: 10.1145/3180155.3180179.

[86] Xie, S.—Khomh, F.—Zou, Y.: An Empirical Study of the Fault-Proneness of Clone Mutation and Clone Migration. Proceedings of the Tenth Working Conference on Mining Software Repositories (MSR), 2013, pp. 149–158, doi: 10.1109/MSR.2013.6624022.

[87] Xue, Y.—Xing, Z.—Jarzabek, S.: CloneDiff: Semantic Differencing of Clones. Proceedings of the 5th International Workshop on Software Clones (IWSC '11), 2011, pp. 83–84, doi: 10.1145/1985404.1985428.

[88] Xuyang, Y.—Chiba, S.: Attempts on Applying Graph Neural Network to Cross-Language Code-Clone Detection. Graduate School of Information Science and Technology, University of Tokyo, 2020. `http://jssst.or.jp/files/user/taikai/2020/FOSE/fose1-3.pdf`.

[89] Yang, W.: Identifying Syntactic Differences Between Two Programs. Software: Practice and Experience, Vol. 21, 1991, No. 7, pp. 739–755, doi: 10.1002/spe.4380210706.

Chaman Wijesiriwardana received his B.Sc. (Hons) special degree in computer science from the University of Peradeniya, Sri Lanka and obtained his M.Sc. in information and communications technology from the Asian Institute of Technology, Thailand and his Ph.D. degree in software engineering from the University of Colombo School of Computing. He worked as Research Assistant in the Software Evolution and Architecture Lab at the University of Zurich for 3 years. His research interests include software evolution analysis, mining software repositories and software security.

Prasad Wimalaratne obtained his B.Sc. special degree in computer science from the University of Colombo and his Ph.D. in virtual environments from the University of Salford, United Kingdom, in 2002. He is Senior Member of IEEE and Member of Computer Society of Sri Lanka (CSSL). He has won several awards including the Presidential Award, University of Colombo Vice Chancellor's Award for Research Excellence, University of Colombo Senate (Open) Awards and CSSL's ICT Researcher of the Year award for research excellence. His research interests include interactive 3D interfaces, unmanned aerial vehicles (UAVs), virtual environments, assistive technology and code analysis. He joined the academic staff of the University of Colombo in 1995 and is currently the Head of the Department of Communication and Media Technologies at the University of Colombo School of Computing.