PRECINCT: An Incremental Approach for Preventing Clone Insertion at Commit Time

Mathieu Nayrolles,
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montreal, Canada
m_nayrol@ece.concordia.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montreal, Canada
abdelw@ece.concordia.ca

***Abstract*—Software clones are considered harmful since they may cause the same buggy code to appear in multiple places in the code, making software maintenance and evolution tasks challenging. Clone detection has been an active research field for almost two decades. Interestingly, most existing techniques focus on detecting clones after they are inserted in the code. In this paper, we take another look at the clone detection problem by designing a novel approach for preventing the insertion of clones in the first place. Our approach, called PRECINCT (PREventing Clones INsertion at Commit Time), detects efficiently type 3 software clones at commit time by means of pre-commit hooks. This way, changes to the code are analyzed and suspicious copies are flagged before they reach the central code repository in the version control system. The application of PRECINCT to three systems developed independently shows that PRECINCT would have prevented 97.7% of clone insertion.**

I. Introduction

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of code in a given software system [1], [2]. Developers often reuse code (and create clones) in their software on purpose [3]. Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in the code [4]–[6]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused in order to fix the bug in each of them.

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into three categories. The first category includes techniques that treat the source code as text and use transformation and normalization to compare various code fragments [7]–[10]. The second category includes methods that use lexical analysis, where the source code is sliced into sequences of tokens, similar to the way a compiler operates [1], [6], [11]–[13]. The tokens are used to compare code fragments. Finally, syntactic analysis has also been performed where the source code is converted into trees, more particularly abstract syntax tree (AST), and then the clone detection is performed using tree matching algorithms [14]–[17].

Despite the advances in clone detection research, the use of existing clone detection tools is not as widespread as one might think. The main factors that contribute to this are summarized in [18]. These tools are known to output a large number of data, making it hard to understand and analyze their results. In addition, they tend to have a high amount of false positive. Finally, they are hard to configure and do not integrate well with the day-to-day workflow of a developer.

In this paper, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones in the first place at commit time, i.e., before they reach the central code repository. PRECINCT uses pre-commit hooks capabilities of modern source code version control systems. A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. In fact, PRECINCT, itself, can be seen as a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code. This said, only a fraction of the code is analyzed, making PRECINCT efficient compared to leading clone detection techniques such as NICAD (Accurate Detection of Near-miss Intentional Clones) [9]. Moreover, the detected clones are presented using a classical 'diff' output that developers are familiar with. PRECINCT is also well integrated with the workflow of the developers since it is used in conjunction with a source code version control systems such as Git and SVN.

Many taxonomies have been published in an attempt to classify clones into types. [19]–[25]. Despite the particularities of each proposed taxonomy, researchers agree to the following classification. Type 1 clones are copy-pasted blocks of code that only differ from each other in terms of non-code artefacts such as indentation, whitespaces, comments and so on. Type 2 clones are blocks of code that are syntactically identical at the exception of literals, identifiers and types that can be modified. In addition, Type 2 also shares the particularities of Type 1 about indentation, whitespaces and comments. Type 3 clones are similar to Type 2 clones in terms of modification of literals, identifiers, types, indentation, whitespaces and comments but

also contain added or deleted code statements. Finally, Type 4 are code blocks that perform the same tasks, but using a completely different implementation.

In this study, we focus on Type 3 clones as they are more challenging to detect. Since Type 3 clones include Type 1 and 2 clones, then these types could be easily detected by PRECINCT as well.

We evaluated the effectiveness of PRECINCT using precision and recall on three systems, developed independently and written in both C and Java. The results show that PRECINCT prevents near-miss software clones to reach the source version system with an average accuracy of 97.7%.

The rest of this paper is organized as follows: In Section II, we present the studies related to PRECINCT. Then, in Section III, we present the PRECINCT approach. The evaluation of PRECINCT is the subject of Section IV. Finally, we propose concluding remarks in Section VI.

II. Related Work

Clone detection is an important and difficult task. Throughout the years, researchers and practitioners have developed a considerable number of methods and tools in order to detect efficiently source code clones.

Text-based techniques use the code — often raw (e.g. with comments) — and compare sequences of code (blocks) to each other in order to identify potential clones. Johnson was perhaps the first one to use fingerprints to detect clones [7], [8]. Blocks of code are hashed; producing fingerprints that can be compared. If two blocks share the same fingerprint, they are considered as clones. Manber et al. [26] and Ducasse et al. [27] refined the fingerprint technique by using leading keywords and dot-plots, respectively.

Tree-matching and metrics-based are two sub-categories of syntactic analysis for clone detection. Syntactic analysis consists of building abstract syntax trees (AST) and analyze them with a set of dedicated metrics or searching for identical sub-trees. Many approaches using AST have been published using sub-trees comparison including the work of Baxter et al. [14], Wahleret al. [28], or more recently, the work of Jian et al. With Deckard [29]. AST-based approach compares metrics computed on the AST, rather than the code itself, to identify clones [30], [31].

Another approach to detect clones is to use static analysis and to leverage the semantics of the program to improve the detection. These techniques rely on program dependency graphs where nodes are statements and edges are dependencies. Then, the problem of finding clones is reduced to the problem of finding identical sub-groups in the program dependency graph. Examples of recent techniques that fall into this category are the ones presented by Krinke et al. [32] and Gabel et al. [33].

Many clone detection tools have been created using a lexical approach for clone detection. Here, the code is transformed into a series of tokens. If sub-series repeat themselves, it means that a potential clone is in the code. Some popular tools that use this technique include, but not limited to, Dup [1], CCFinder [13], and CP-Miner [6].

Furthermore, a large number of taxonomies have been published in an attempt to classify clones and ease the research on clone detection [19]–[25].

Other active research activities in clone detection focus on clone removal and management. Once detected, an obvious step is to provide approaches to remove clones in an automatic way or (at least) keep track of them if removing them is not an option. Most modern IDEs provide the *extract method* feature that transforms a potentially copy-pasted block of code into a method and a call to the newly generated method [34], [35]. More advanced techniques involve analyzing the output of CCFinder [36] or program dependencies graphs [35] to automatically suggest a method that would go through the *extract method* process Codelink [37] and [38].

The aforementioned techniques, however, focus on detecting clones after they are inserted in the code. A few studies only focus on preventing the insertion of clones in the first place. Lague et al. [39] conducted a very large empirical study with 10,000 developers over 3 years, where developers where asked to use clone detection tools during the development process of a very large telecoms system. The authors found that while clones are being removed over time, using clone detection tools help improving the quality of the system as it prevent defects to reach the customers. Duala et al. [38], [40] proposed to create clone region descriptors (CRDs), which describe clone regions within methods in a robust way that is independent from the exact text of the clone region or its location in a file. Then, using CRDs, clone insertion can be prevented.

PRECINCT aims to prevent clone insertion while integrating the clone detection process in a transparent manner in the day-to-day development process. This way, software developers do not have to resort to external tools to remove clones after they are inserted. Our approach operates at commit time, notifying software developers of possible clones as they commit their code.

III. The PRECINCT Approach

The PRECINCT approach is composed of six steps. The first two steps are part of the developer workflow. Indeed, the first step is the commit step where developers send their latest changes to the central repository and the last step is the reception of the commit by the central repository. The second step is the pre-commit hook which kicks in as the first operation when one wants to commit. The pre-commit hook has access to the changes in terms of files that have been modified, more specifically, the lines that have been modified. The modified lines of the files are sent to TXL [41] for block extraction. Then, the blocks are compared to previously extracted blocks in order to identify candidate clones using NICAD [9]. Finally, the output of NICAD is further refined and presented to the user for a decision round. These steps are discussed in more detail in the following subsections.

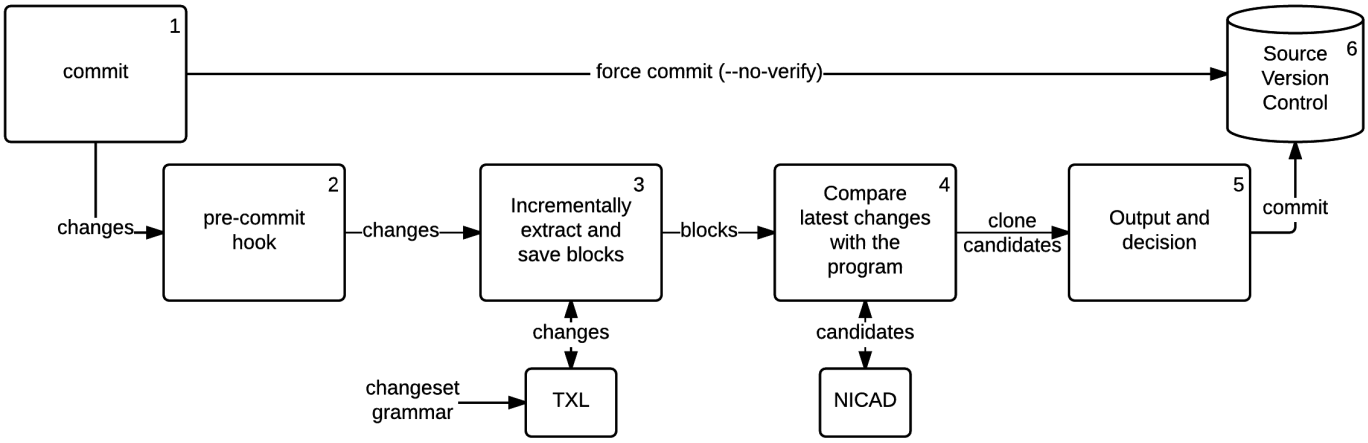

Fig. 1. Overview of the PRECINCT Approach.

A. Commit

In version control systems, a commit adds the latest changes made to the source code to the repository, making these changes part of the head revision of the repository. Commits in version control systems are kept in the repository indefinitely. Thus, when other users do an update or a checkout from the repository, they will receive the latest committed version, unless they wish to retrieve a previous version of the source code in the repository. Version control systems allow rolling back to previous versions easily. In this context, a commit within a version control system is protected as it is easily rolled back, even after the commit has been done.

B. Pre-Commit Hook

Hooks are custom scripts set to fire off when certain important actions occur. There are two groups of hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used for all sorts of reasons such as compliance to coding rules or automatic run of unit test suites.

The pre-commit hook is run first, before one even types in a commit message. It is used to inspect the snapshot that is about to be committed. Depending on the exit status of the hook, the commit will be aborted and not pushed to the central repository. Also, developers can choose to ignore the pre-hook by using `git commit --no-verify` instead of `git commit`. This can be useful in case of emergency bug fix where the code has to reach the central repository as quickly as possible. Developers can do things like check for code style (run lint or something equivalent), check for trailing white spaces (the default hook does exactly this), or check for appropriate documentation on new methods.

PRECINCT is a set of bash scripts where the entry point of these scripts lies in the pre-commit hooks. Pre-commit hooks are easy to create and implement as depicted in Listing 1. This pre-hook is shipped with Git[1], a popular version control system. Note that even though we use Git as the main version control to present PRECINCT, we believe that the techniques presented in this paper are readily applicable to other version control systems. In Listing 1, from lines 3 to 11, the script identifies if the commit is the first one in order to select the revision to work against. Then, in Lines 18 and 19, the script checks for trailing whitespace and fails if any are found.

Listing 1. Git Pre-Commit Hook Sample

```
#!/bin/sh                                          1
                                                   2
if git rev-parse --verify HEAD > \                 3
 /dev/null 2>&1                                    4
then                                               5
 against=HEAD                                      6
else                                               7
 # Initial commit: diff against                    8
 # an empty tree object                            9
 against=4b825dc642....                            10
fi                                                 11
                                                   12
# Redirect output to stderr.                       13
exec 1>&2                                          14
                                                   15
# If there are whitespace errors,                  16
# print the offending file names and fail.         17
exec git diff-index --check \                      18
        --cached $against --                       19
```

For PRECINCT to work, we just have to add the call to our script suite instead or in addition of the whitespace check.

C. Extract and Save Blocks

A block is a set of consecutive lines of code that will be compared to all other blocks in order to identify clones. To achieve this critical part of PRECINCT, we rely on TXL [41], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [41]. For TXL to work, one has to write a grammar describing the syntax of the source language and the transformations needed. TXL has three main

[1]https://git-scm.com/

phases: *parse, transform*, *unparse*. In the parse phase, the grammar controls not only the input but also the output form. Listing 2 — extracted from the official documentation[2] — shows a grammar matching a *if-then-else* statement in C with some special keywords: [IN] (indent), [EX] (exdent) and [NL] (newline) that will be used for the output form.

Listing 2. Txl Sample Sample

```
define if_statement                          1
  if ( [expr] ) [IN][NL]                     2
    [statement] [EX]                         3
    [opt else_statement]                     4
end define                                   5
                                             6
define else_statement                        7
  else [IN][NL]                              8
    [statement] [EX]                         9
end define                                   10
```

Then, the *transform* phase will, as the name suggests, apply transformation rules that can, for example, normalize or abstract the source code. Finally, the third phase of TXL, called *unparse*, unparses the transformed parsed input in order to output it. Also, TXL supports what the creators call Agile Parsing [42], which allow developers to redefine the rules of the grammar and, therefore, apply different rules than the original ones.

PRECINCT takes advantage of that by redefining the blocks that should be extracted for the purpose of clone comparison, leaving out the blocks that are out of scope. More precisely, before each commit, we only extract the blocks belonging to the modified parts of the source code. Hence, we only process, in an incremental manner, the latest modification of the source code instead of the source code as a whole.

We have selected TXL for several reasons. First, TXL is easy to install and to integrate with a developer workflow. Second, it was relatively easy to create a grammar that accepts commits as input. This is because TXL is shipped with C, Java, Csharp, Python and WSDL grammars that define all the particularities of these languages, with the ability to customize these grammar to to accept changesets (chunks of the modified source code that includes the added, modified, and deleted lines) instead of the whole code.

Algorithm 1 presents an overview of the "extract" and "save" blocks operations. This algorithm receives as arguments, the changesets, the blocks that have been previously extracted and a boolean named compare_history. Then, from Lines 1 to 9 lie the for loop that iterates over the changesets. For each changeset (Line 2), we extract the blocks by calling the $extract\_blocks(Changeset\ cs)$ function. In this function, we expand our changeset to the left and to the right in order to have a complete block. As depicted by Listing 3, changesets contain only the modified chunk of code and not necessarily complete blocks. Indeed, we have a block from Line 3 to Line 6 and deleted lines from Line 8 to 14. However, in Line 7 we can see the end of a block, but we do not have its beginning. Therefore, we need to expand the changeset to the left in order to have syntactically correct blocks. We do so by checking the block's beginning and ending, { and } in C for example. Then, we send these expanded changesets to TXL for block extraction and formalization.

[2]http://txl.ca

```
   Data: Changeset[] changesets;
   Block[] prior_blocks;
   Boolean compare_history;
   Result: Up to date blocks of the systems
1  for i ← 0 to size_of changesets do
2  |  Block[] blocks ← extract_blocks(changesets);
3  |  for j ← 0 to size_of blocks do
4  |  |  if not compare_history AND blocks[j] overrides
   |  |  one of prior_blocks then
5  |  |  |  delete prior_block;
6  |  |  end
7  |  |  write blocks[j];
8  |  end
9  end
10 Function extract_blocks(Changeset cs)
11 |  if cs is unbalanced right then
12 |  |  cs ← expand_left(cs);
13 |  else if cs is unbalanced left then
14 |  |  cs ← expand_right(cs);
15 |  end
17 |  return txl_extract_blocks(cs);
```

Algorithm 1: Overview of the Extract Blocks Operation

Listing 3. Changeset c4016c of monit

```
@@ -315,36 +315,6 @@                                  1
 int initprocesstree_sysdep                          2
     (ProcessTree_T **reference) {                   3
         mach_port_deallocate(mytask,                4
           task);                                    5
     }                                               6
 }                                                   7
-  if (task_for_pid(mytask, pt[i].pid,               8
-      &task) == KERN_SUCCESS) {                     9
-    mach_msg_type_number_t    count;                10
-    task_basic_info_data_t                          11
taskinfo;
-    thread_array_t                                  12
threadtable;
-    unsigned int                                    13
threadtable_size;
-    thread_basic_info_t                             14
threadinfo;
```

For each extracted block, we check if the current block overrides (replaces) a previous block (Line 4). In such a case, we delete the previous block as it does not represent the current version of the program anymore (Line 5). Also, we have an optional step in PRECINCT defined in Line 4. The compare_history is a condition to delete overridden blocks.

We believe that deleted blocks have been deleted for a good reason (bug, default, removed features, ...) and if a newly inserted block matches an old one, it could be worth knowing in order to improve the quality of the system at hand. This feature is deactivated by default.

In summary, this step receives the files and lines, modified by the latest changes made by the developer and produces an up to date block representation of the system at hand in an incremental way. The blocks can be analyzed in the next step to discover potential clones.

D. Compare Extracted Blocks

In order to compare the extracted blocks and detect potential clones we can only resort to text-based techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, a program that compiles. In the relatively wide-range of tool and techniques that exist to detect clones by considering code as text [2], [7], [8], [26], [43], [44], we selected NICAD as the main text-based method for comparing clones [9] for several reasons. First, NICAD is built on top of TXL, which we also used in the previous step. Second, NICAD is able to detect all type 1, 2 and 3 software clones. While in this study, we focus on type 3, as explained in I, this capacity will be used for future studies.

NICAD works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase all potential clones are identified, pretty-printed, and extracted. We do not use the *Extraction* phase of NICAD as it has been built to work on programs that are syntactically correct, which is not the case of changesets. We replaced NICAD's *Extraction* phase of our own tool, described in the previous section.

In the *Comparison* phase, extracted blocks are transformed, clustered and compared in order to find potential clones. Using TXL sub-programs, blocks go through a process called pretty-printing where they are stripped of formatting and comments. When code fragments are cloned, some comments, indentation or spacing are changed according to the new context where the new code is used. This pretty-printing process ensures that all code will have the same spacing and formatting and ease the comparison. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table I shows how this can improve the accuracy of clone detection with three `for` statements, `for (i=0; i<10; i++)`, `for (i=1; i<10; i++)` and `for (j=2; j<100; j++)`. The pretty-printing allows NICAD to detect Segments 1 and 2 as a clone pair because only the initialization of i changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc [27]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

Finally, the extracted, pretty-printed, normalized and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [46]. Then, a percentage of unique statements can be computed and, depending on a given threshold (see Section IV), the blocks are marked as clones.

TABLE I
PRETTY-PRINTING EXAMPLE [45]

Segment 1	Segment 2	Segment 3	S1 & S2	S1 & S3	S2 & S3
for (	for (	for (	1	1	1
i = 0;	i = 1;	j = 2;	0	0	0
i >10;	i >10;	j >100;	1	0	0
i++)	i++)	j++)	1	0	0
Total Matches			3	1	1
Total Mismatches			1	3	3

The last step of NICAD, which acts as our clone comparison engine, is the *reporting*. However, to prevent PRECINCT from outputting a large number of data (an issue from which many clone detection techniques face), we implemented our own reporting system, which is also well embedded with the workflow of developers. This reporting system is the subject of the next section.

As a summary, this step receives potentially expanded and balanced blocks from the extraction step. Then, the blocks are pretty-printed, normalized, filtered and fed to an LCS algorithm in order to detect potential clones. Moreover, the clone detection in PRECINCT is less intensive than NICAD as we only compare the latest changes with rest of the program instead of comparing all the blocks with each other.

E. Output and Decision

In this final step, we report the result of the clone detection at commit time with respect to the latest changes made by the developer. The process is straightforward. Every change made by the developers has been through the previous steps and might have been marked as a potential clone. For each file that is suspected to contain a clone, one line is printed to the command line with the following options: (I) Inspect, (D) Disregard, (R) Remove from the commit as shown by Figure 2. In comparison to this simple and interactive output, NICAD outputs each and every details of the detection such as total of potential clones, total of lines, total of unique line text chars, total unique lines, total comparisons and so on. Then the potential clones are stored in XML files that can be viewed using an Internet browser or a text editor.

(I) Inspect will cause a diff-like visualization of the suspected clones while (D) disregard will simply ignore the finding. To integrate PRECINCT in the workflow of the developer we also propose the remove option (R). This option will simply remove the suspected file from the commit that is about to be sent to the central repository. Also, if the user types an option key twice, e.g. II, DD or RR, then, the option will be applied to all files. For instance, if the developer types DD at any one point, the PRECINCT's results will be disregarded and the commit will be allowed to go through. We believe that this simple mechanism will encourage developers to use PRECINCT like they would use any other feature of Git (or any other control version system).

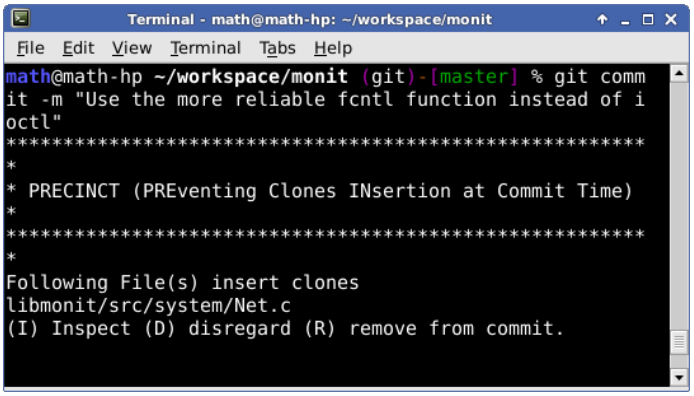

Fig. 2. PRECINCT output when replaying commit `710b6b4` of monit.

TABLE II
LIST OF TARGET SYSTEMS IN TERMS OF FILES AND KILO LINE OF CODE (KLOC) AT CURRENT VERSION AND LANGUAGE

SUT	Revisions	Files	KLoC	Language
Monit	826	264	107	C
Jhotdraw	735	1984	44	Java
dnsjava	1637	233	47	Java

IV. EXPERIMENTATIONS

In this section, we show the effectiveness of PRECINCT to detect clones at commit time in three open source systems[3].

The aim of the case study is to answer the following question: *Can we detect insertion clones at commit time, i.e., before they are inserted in the final code, if so, what would be the accuracy?*

A. Target Systems

Table II shows the systems used in this study and their characteristics in terms of the number files they contain and the size in KLoC (Kilo Lines of Code). We also include the number of revisions used for each system and the programming language in which the system is written.

Monit[4] is a small open source utility for managing and monitoring Unix systems. Monit is used to conduct automatic maintenance and repair and supports the ability to identify causal actions to detect errors. This system is written in C and composed of 826 revisions, 264 files, and the latest version has 107 KLoC. We have chosen Monit as a target system because it was one of the systems NICAD was tested on.

JHotDraw[5] is a Java GUI framework for technical and structured Graphics. It has been developed as a "design exercise". Its design relies heavily on the use of design patterns. JHotDraw is composed of 735 revisions, 1984 files, and the latest revision has 44 KLoC.It is written and Java and it is often used by researchers as a test bench. JHotDraw was also used by NICAD's developers to evaluate their approach.

Dnsjava[6] is a tool for implementing the DNS (Domain Name Service) mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it has been well maintained for the past decade. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab. Dnsjava is composed of 1637 revisions, 233 files and 47 KLoC at the latest revision. We have chosen this system because we are familiar with it as we used it before [47].

[3]The programs used and instructions to reproduce the experiments are made available for download from https://research.mathieu-nayrolles.com/precinct/

[4]https://mmonit.com/monit/

[5]http://www.jhotdraw.org/

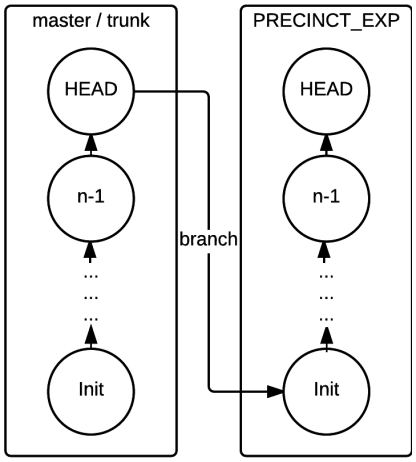

Fig. 3. PRECINCT Branching.

B. Process

Figure 3 shows the process we followed to validate the effectiveness of PRECINCT.

As our approach relies on commit pre-hooks to detect possible clones during the development process (more particularly at commit time), we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *PRECINCT_EXT*. When created, this branch is reinitialized at the initial state of the project (the first commit) and each commit can be replayed as they have originally been. At each commit, we store the time taken for PRECINCT to run as well as the number of detected clone pairs. We also store the size of the output in terms of number of lines of code, output by our method.

To validate the results obtained by PRECINCT, we needed to use a reliable clone detection approach to which we compare our results. For this, we use NICAD because of its popularity, high accuracy, and availability [9]. This means, we run NICAD on the revisions of the system to obtain the clones. We use NICAD clones as a baseline for comparing the results obtained by PRECINCT.

We show the result of detecting Type 3 clones with a maximum line difference of 30% as discussed in Table I. As

[6]http://www.dnsjava.org/

discussed in the introductory section, we chose to focus on Type 3 clones because they are more challenging to detect than Type 1 and 2. PRECINCT detects Type 1 and 2 too so does NICAD. For the time being, PRECINCT is not designed to detect Type 4 clones. These clones use different implementations. Detecting Type 4 clones is part of future work.

We assess the performance of PRECINCT in terms of precision (Equation 1) and recall (Equation 2). Both the precision and the recall are computed using NICAD's results as a baseline. We also compute F_1-measure (Equation 3), i.e., the weighted average of precision and recall, to measure the accuracy of PRECINCT.

$$precision = \frac{|\{NICAD_{detection}\} \cap \{PRECINCT_{detection}\}|}{|\{PRECINCT_{detection}\}|} \quad (1)$$

$$recall = \frac{|\{NICAD_{detection}\} \cap \{PRECINCT_{detection}\}|}{|\{NICAD_{detection}\}|} \quad (2)$$

$$F_1 - measure = 2 * \frac{precision * recall}{precision + recall} \quad (3)$$

C. Results

Figures 4, 5, 6 show the results of our study in terms of clone pairs that are detected per revision for our three subject systems: Monit, JHotDraw and Dnsjava. We compared our approach to NICAD that detects the clones after they are inserted. The blue line shows the clone detection performed by NICAD, while the red line shows the clone pairs that have been missed by PRECINCT (i.e., the ones that would reach the central code repository).

Table III summarizes PRECINCT's results in terms of precision, recall, F_1-measure, execution time and output reduction. The first version of Monit contains 85 clone pairs and this number stays stable until Revision 100. From Revision 100 to 472 the detected clone pairs vary between 68 and 88 before reaching 219 at Revision 473. The number of clone pairs goes down to 122 at Revision 491 and decreases to 128 in the last revision. PRECINCT was able to detect 96.1% (123/128) of the clone pairs that are detected by NICAD with a 100% recall. It took in average around 1 second for PRECINCT to execute on a Debian 8 system with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8Gb of DDR3 memory. It is also worth mentioning that the computer we used is equipped with SSD (Solid State Drive). This impacts the running time as clone detection is a file intensive operation. Finally, the PRECINCT was able to output 88.3% less lines of code than NICAD.

JHotDraw starts with 196 clone pairs at Revision 1 and reaches a pick of 2048 at Revision 180. The number of clones continues to go up until Revisions 685 and 686 where the number of pairs is 1229 before picking at 6538 and more from Revisions 687 to 721. PRECINCT was able to detect 98.3% of the clone pairs detected by NICAD (6490/6599) with 100% recall while executing on average in 1.7 second (compared to 5.1 seconds for NICAD). With JHotDraw, we can clearly see the advantages of incremental approaches. Indeed, the execution time of PRECINCT is loosely impacted by the number of files inside the system as the blocks are constructed incrementally. Also, we only compare the latest change to the remaining of the program and not all the blocks to each other as NICAD. We also were able to reduce by 70.1% the number of lines output by NICAD.

Finally, for Dnsjava, the number of clone pairs starts high with 258 clones and goes up until Revision 70 where it reaches 165. Another quick drop is observed at Revision 239 where we found only 25 clone pairs. The number of clone pairs stays stable until Revision 1030 where it reaches 273. PRECINCT was able to detect 82.8% of the clone pairs detected by NICAD (226/273) with 100% recall, while executing on average in 1.1 second while NICAD took 3 seconds in average. PRECINCT outputs 83.4% less lines of code than NICAD.

Overall, PRECINCT prevented 97.7% of the 7000 clones to reach the central source code repository while executing more than twice as fast as NICAD (1.2 Sec versus 3.0 Sec in average) while reducing the output in terms of lines of code by 83.4% in average.

The difference in execution time between NICAD and PRECINCT stems from the fact that, unlike PRICINCT, NICAD is not an incremental approach. For each revision, NICAD has to extract all the code blocks and then compares all the pairs with each other (n^2). On the other hand, PRECINCT only extracts blocks when they are modified and only compares what has been modified in the rest of the program.

The difference in precision between NICAD and PRECINCT (2.3%) can be explained by the fact that sometimes developers commit code that does not compile. Such commits will still count as a revision, but TXL fails to extract blocks that do not comply with the target language syntax. While NICAD also fails in such a case, the disadvantage of PRECINCT comes from the fact that the failed block is saved and used as reference until it is changed by a correct one in another commit.

V. Threats to Validity

The selection of systems under tests (SUTs) is one of the common threats to validity for approaches aiming to improve the understanding of program's behavior. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the SUTs analysed by PRECINCT are the same as the ones used in similar studies. Moreover, the SUTs vary in terms of purpose, size and history.

Another threat to validity lies in the way we have selected the versions used in this study. We selected version randomly to avoid any bias. One may argue that a better approach would be to select a revision based on size or diversity. However, we believe that our approach does not depend on

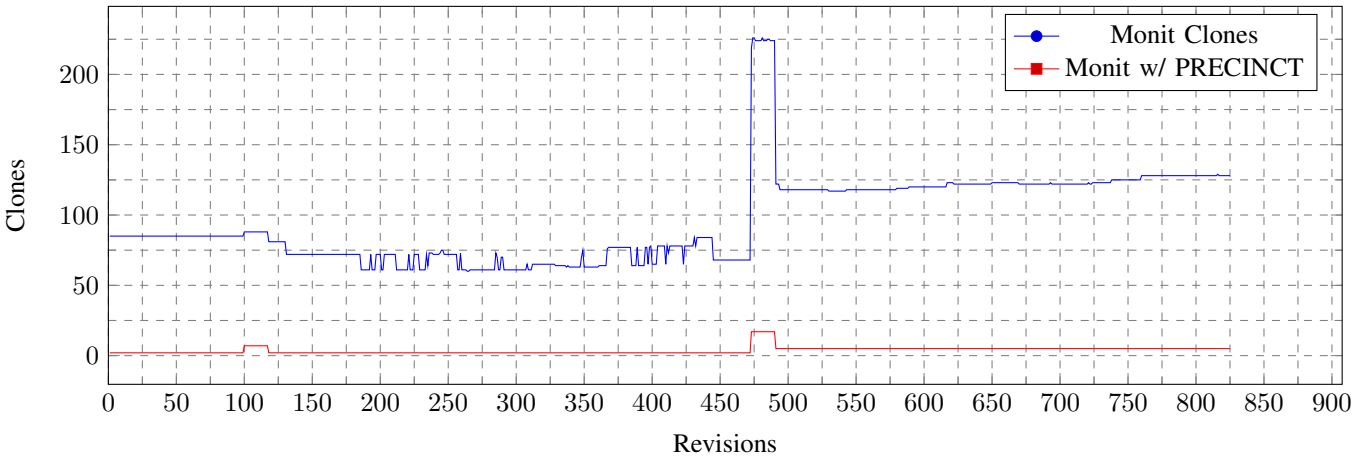

Fig. 4. Monit Clone evolution over revisions

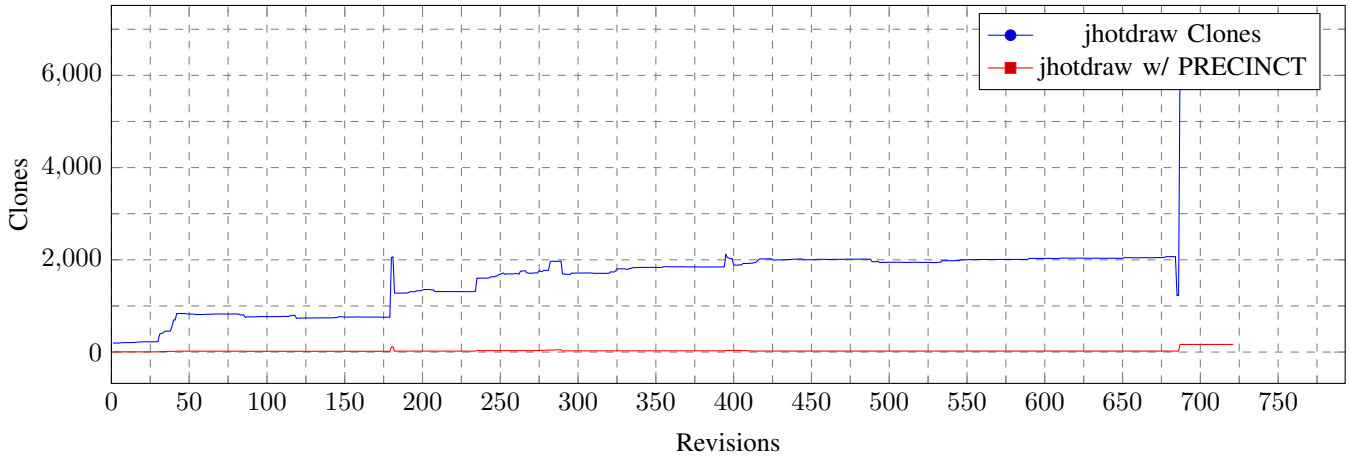

Fig. 5. jhotdraw Clone evolution over revisions

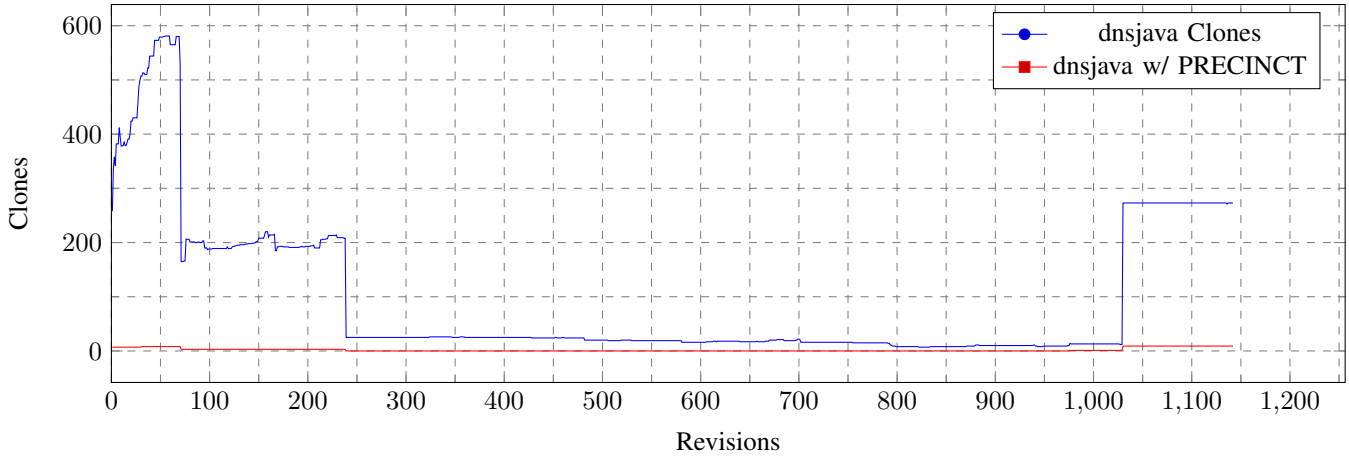

Fig. 6. dnsjava Clone evolution over revisions

TABLE III
OVERVIEW OF PRECINCT'S RESULTS IN TERMS OF PRECISION, RECALL, F_1-MEASURE, EXECUTION TIME AND OUTPUT REDUCTION.

	NICAD	PRECINCT	Precision	Recall	F1-measure	NICAD's Average Execution Time	PRECINCT's Average Execution Time	Overall Output Reduction
Monit	128	123	96.1%	100%	98%	2.2s	0.9s	88.3%
JHotDraw	6599	6490	98.3%	100%	99.1%	5.1s	1.7s	70.1%
DnsJava	273	226	82.8%	100%	90.6%	1.8s	1.1s	88.6%
Total	7000	6839	97.7%	100%	98.8%	3s	1.2s	83.4%

the characteristics of the revision, but on the characteristics of the SUT.

In addition, we see a threat to validity that stems from the fact that we only used open source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

The programs we used in this study are all based on the Java, C and Python programming languages. This can limit the generalization of the results. However, similar to Java, C, Python, if one writes a TXL grammar for a new language — which can be a relatively hard work — then PRECINCT can work since PRECINT relies on TXL.

Finally, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of PRECINCT. This said, since NICAD has been tested on large systems, we are confident that it is a suitable engine for comparing code using TXL. Also, there is nothing that prevents from using other code comparisons engines, if need be.

In conclusion, internal and external validity have both been minimized by choosing a set of three different systems, using input data that can be found in any programming languages and version systems (commit and changesets).

VI. CONCLUSION

We presented PRECINCT (PREventing Clones INsertion at Commit Time), an incremental approach for preventing clone insertion at commit time that combines efficient block extraction and clone detection and integrate itself seamlessly in the day-to-day workflow of developers. PRECINCT takes advantage of TXL and NICAD to create a clone detection tool approach that runs automatically before each commit in 1.2 second with a 97.7% precision and a 100% recall (when using NICAD results as a baseline).

Our approach also assesses two major factors that contribute to the slow adoption of clone detection tools: large number of data output by clone detection methods, and smooth integration with the task flow of the developers. PRECINCT is able to reduce the number of lines output by a classical clone detection tool such as NICAD by 83.4% while keeping all the necessary information that allow developers to decide whether the detect clone is in fact a clone. Also, our approach is seamlessly integrated with the developers' workflow by means of pre-commit hooks, which are part any version control systems.

To build on this work, we need to experiment with additional (and larger) systems with the dual aim to (a) improve and fine-tune the approach, and (b) assess the scalability of our approach when applied to even larger (and proprietary) systems. Also, we want to improve PRECINCT to support Type 4 clones.

REFERENCES

[1] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE Comput. Soc. Press, pp. 86–95. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=514697

[2] S. D. Stéphane Ducasse, Matthias Rieger, "A Language Independent Approach for Detecting Duplicated Code." [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.6060

[3] M. Kim, V. Sazawal, and D. Notkin, "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 187, sep 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1095430.1081737

[4] C. Kapser and M. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in *2006 13th Working Conference on Reverse Engineering*. IEEE, oct 2006, pp. 19–28. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4023973

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, may 2009, pp. 485–495. [Online]. Available: http://dl.acm.org/citation.cfm?id=1555001.1555062

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, mar 2006. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1610609

[7] J. H. Johnson, "Visualizing textual redundancy in legacy source," p. 32, oct 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=782185.782217

[8] ——, "Identifying redundancy in source code using fingerprints," pp. 171–183, oct 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=962289.962305

[9] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, jun 2011, pp. 219–220. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5970189

[10] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, jun 2008, pp. 172–181. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4556129

[11] B. S. Baker, "A program for identifying duplicated code." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.550.4540

[12] B. S. Baker and R. Giancarlo, "Sparse Dynamic Programming for Longest Common Subsequence from Fragments," *Journal of Algorithms*, vol. 42, no. 2, pp. 231–254, feb 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677402912149

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, jul 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=636188.636191

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," p. 368, mar 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[15] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '00*. New York, New York, USA: ACM Press, jan 2000, pp. 155–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=325694.325713

[16] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proceedings of the 44th annual southeast regional conference on - ACM-SE 44*. New York, New York, USA: ACM Press, mar 2006, p. 679. [Online]. Available: http://dl.acm.org/citation.cfm?id=1185448.1185597

[17] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, jul 2008. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9073-9

[18] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" pp. 672–681, may 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[19] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance ICSM-96*. IEEE, 1996, pp. 244–253. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=565012

[20] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*. IEEE Comput. Soc, 1999, pp. 292–303. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=809750

[21] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, oct 2006, pp. 253–262. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4023995

[22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, sep 2007. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4288192

[23] S. F. R. J. F. Neil Davey, Paul Barson, "The Development of a Software Clone Detector." [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7548

[24] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE Comput. Soc, pp. 44–54. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=624575

[25] C. Kapser and M. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE, pp. 85–94. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1334772

[26] U. Manber, "Finding similar files in a large file system," p. 2, jan 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267076

[27] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. IEEE, 1999, pp. 109–118. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=792593

[28] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE Comput. Soc, pp. 128–135. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1386166

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 96–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=1248820.1248843

[30] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, "Extending software quality assessment techniques to Java systems," in *Proceedings Seventh International Workshop on Program Comprehension*. IEEE Comput. Soc, 1999, pp. 49–56. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=777743

[31] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. IEEE Comput. Soc, pp. 326–336. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=806971

[32] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," p. 301, oct 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=832308.837142

[33] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, 2008, p. 321. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4814143

[34] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*. IEEE Comput. Soc, pp. 33–42. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1199187

[35] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis," in *Product Focused Software Process Improvement*. Springer, 2004, pp. 220–233.

[36] F. Bomarius and H. Iida, Eds., *Product Focused Software Process Improvement*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3009. [Online]. Available: http://www.springerlink.com/index/10.1007/b96726

[37] M. Toomim, A. Begel, and S. Graham, "Managing Duplicated Code with Linked Editing," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. IEEE, pp. 173–180. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1372317

[38] E. Duala-Ekoko and M. P. Robillard, "Tracking Code Clones in Evolving Software," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 158–167. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4222578

[39] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proceedings International Conference on Software Maintenance*. IEEE Comput. Soc, pp. 314–321. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5726968

[40] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, pp. 1–31, jun 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1767751.1767754

[41] J. R. Cordy, "Source transformation, analysis and generation in TXL," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '06*. New York, New York, USA: ACM Press, jan 2006, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1111542.1111544

[42] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," *Automated Software Engineering*, vol. 10, no. 4, pp. 311–336. [Online]. Available: http://link.springer.com/article/10.1023/A%3A1025801405075

[43] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc, pp. 107–114. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=989796

[44] R. Wettel and R. Marinescu, "Archeology of code duplication: recovering duplication chains from small duplication fragments," in *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 2005, p. 8 pp. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1595830

[45] N. E. A. R. Iss and S. O. C. Lones, "D Etection and a Nalysis of," no. August, 2009.

[46] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, may 1977. [Online]. Available: http://dl.acm.org/citation.cfm?id=359581.359603

[47] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "JCHARMING: A bug reproduction approach using crash traces and directed model checking," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 101–110, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7081820