SPCP-Miner: A Tool for Mining Code Clones That Are Important for Refactoring or Tracking

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—Code cloning has both positive and negative impacts on software maintenance and evolution. Focusing on the issues related to code cloning, researchers suggest to manage code clones through refactoring and tracking. However, it is impractical to refactor or track all clones in a software system. Thus, it is essential to identify which clones are important for refactoring and also, which clones are important for tracking. In this paper, we present a tool called *SPCP-Miner* which is the pioneer one to automatically identify and rank the important refactoring as well as important tracking candidates from the whole set of clones in a software system. *SPCP-Miner* implements the existing techniques that we used to conduct a large scale empirical study on *SPCP clones* (i.e., the clones that evolved following a *Similarity Preserving Change Pattern* called *SPCP*). We believe that *SPCP-Miner* can help us in better management of code clones by suggesting important clones for refactoring or tracking.**

I. Introduction

If two or more code fragments in a code-base are exactly or nearly similarly to one another, we call them code clones. Two or more similar code fragments form a clone class. Code clones are mainly created because of the frequent copy/paste activities of the programmers during software development and maintenance. Clones are of great importance from the perspectives of software maintenance and evolution [1], [8], [9], [11], [18]. Although cloning helps us in faster software development and program comprehension, there are strong empirical evidences [1], [11] of some negative effects of code cloning such as - hidden bug propagation [1], unintentional inconsistent changes [1], higher instability [11], and late propagation [1]. Focusing on the issues related to code clones researchers suggest to manage clones through refactoring and tracking.

Clone refactoring refers to the technique of merging two or more clone fragments into a single one without altering system behaviour. Clone tracking means remembering the clone fragments in a particular clone class so that while changing a particular clone fragment from that class in future, we can also look at the other clone fragments in the class to decide whether we also need to make similar changes to these other fragments to maintain consistency.

Before applying clone management activities (such as refactoring or tracking) it is important to identify which clones are important for refactoring or tracking. There are empirical evidences [9] that it is impractical to aggressively refactor all clones in a software system, because some clones are volatile. Also, a significant amount of clones do not change at all during evolution, and a considerable amount of clones have the possibility of evolving independently [9] . These clones should neither be considered for refactoring or nor for tracking. Moreover, sometimes removal of clone fragments through refactoring might negatively affect the future evolution of related non-clone fragments [12]. Such clone fragments should be considered for tracking rather than refactoring. We should also note that the task of clone refactoring is often time consuming and costly because it requires interaction of experienced programmers and also, after refactoring the modified code snippets might be subject to testing whether they are performing their expected functionalities. Thus, identification of important clones for refactoring as well as for tracking is important. If the programmers do not have any knowledge of which clones are important to be refactored or tracked, they might get overwhelmed while taking clone management decisions from a large set of clones of a software system.

The existing clone refactoring or scheduling tools [2], [5], [10], [16], [18] only focus on identifying refactorable clones by analyzing their structural issues. However, two clone fragments might be refactorable on the basis of their syntactic structure but they might not be important to be refactored on the basis of their past evolution history. These existing refactoring techniques and tools cannot determine which clones are important to be refactored. Higo and Kusumoto [6] performed a small scale study on identifying refactoring candidates on the basis of their past evolution history. According to their consideration if two or more clone fragments received the same changes during the past evolution, they are important candidates for refactoring. However, two clone fragments might not co-evolve by always receiving the same changes, but they can still be important candidates for refactoring. Late propagation clone pairs can be considered as examples of such cases. During late propagation two clone fragments might evolve independently for a period before getting re-synchronized. Such clone fragments should be considered for refactoring because they have the tendency of preserving their similarity. Higo and Kusumoto's approach exclude these clones from refactoring considerations. Moreover, their approach cannot identify important clones for tracking. A number of clone tracking techniques and tools [4], [7], [17] are also available. However, none of these can determine which clones are important for tracking.

Focusing on the above issues, in this paper we present a tool called *SPCP-Miner* [15] which is capable of performing two tasks: **(1)** *automatic identification of the important clones for refactoring and also, the important clones for tracking from a large set of clones reported by the clone detectors*, and **(2)** *prioritizing these important clones according to the necessity of refactoring as well as tracking*. Thus, *SPCP-Miner* can complement the existing clone detection, refactoring, and tracking tools. *SPCP-Miner* implements the existing techniques that we used for performing a large scale empirical study [12] regarding the identification and ranking of important clones for refactoring or tracking. So far as we have studied, there are no other existing tools that can automatically identify clones that are important for refactoring, or for tracking.

We name our tool (*SPCP-Miner*) focusing on the pattern, SPCP (*Similarity Preserving Change Pattern*), that it mines by analyzing the clone evolution history of a software system. *SPCP-Miner* detects all those clones that evolved following an SPCP. We refer these detected clones as *SPCP clones*. SPCP clones are important from the perspectives of clone management (such as clone refactoring and tracking) [12]. The non-SPCP clone fragments either evolve independently or are rarely changed during evolution and thus, should not be considered important for management [12], [14]. *SPCP-Miner* also separates the SPCP clones into two disjoint subsets by analyzing their evolutionary coupling. One set contains those clones that are important for refactoring and the other set contains clones that are important for tracking. According to our previous empirical study [12] on thousands of revisions of six diverse subject systems written in two different programming languages, around 23.47% of all clones in a software sytem can be SPCP clones. We also reported that 13.20% of all clones can be important for refactoring, and the remaining 10.27% (= 23.47 - 13.20) clones can be important for tracking. *SPCP-Miner* not only detects all such clones but also ranks them on the basis of the necessity of refactoring and tracking [12]. Here, we should note that in a previous study [14] we implemented a tool called *MARC* for detecting *SPCP clones*. However, *MARC* does not have the capability of determining which of these clones are important for tracking or refactoring. *SPCP-Miner* can do this by analyzing the evolutionary coupling of the SPCP clones. Also, the SPCP clone detection strategy of *SPCP-Miner* is significantly improved than that of *MARC*.

II. An Example Use Case

Let us assume that a software system has been in the maintenance phase for a long time without clone management. Recently, the client reported a bug saying that certain functionality in a certain module of the software system is not working in the expected way. A developer was appointed to fix the bug. After investigating the bug in the reported module she fixes it by modifying a piece of code in that module.

However, after some days the client again reports a similar bug in another module. The project manager asks the developer to fix it and investigate why the same bug is being reported even after fixing. After investigating the programmer reports that there was a similar piece of buggy code in the newly reported module. She also fixed it in the similar way. Then, the project manager suspects that the similar piece of buggy code might even exist in some other places not yet reported. So, he asks the developer to search for those in the whole code-base. The developer investigates and finds that some other similar pieces of code with the same bug really exist in the code-base and these code fragments also need to be fixed.

From this the project manager realizes that the code-base might have many other groups of similar code fragments. He feels the necessity of refactoring each of these groups so that in order to fix such a bug in future a developer does not need to change in too many places. He asks the developer to at first detect code clones in the software system using a clone detection tool and then, to refactor those clones.

The developer downloads the clone detection and refactoring tools. She applies a clone detector and detects a large amount of clones grouped into different clone groups from the system. Then, she attempts to apply a refactoring tool on the detected clones. However, she experiences that there are many situations where the refactoring tools cannot refactor the clone fragments. She can manually refactor in some of these situations. Also, during manual checking she understands that many clones might not be important to be refactored and also many clones are not even refactorable. Moreover, while deciding to refactor some clones she feels the necessity of understanding how they evolved in the past. She realizes that she should primarily focus on those clones that are more change-prone and have a tendency of co-evolving preserving their similarity. The clones that never changed in the past might be kept in the system as they are because they have very low probability of getting changed in future. She also understands that some clone fragments are eligible to be refactored on the basis of their syntactic structure however, they are closely related to their surrounding non-clone fragments, and thus removal of these clone fragments through refactoring might not be a wise decision. Removal of such clone fragments might negatively affect the future evolution of the related non-clone fragments. Such clone fragments might be important for tracking using a clone tracker. However, to consider all these things she needs to analyze the clone evolution history. She understands that deciding important refactoring as well as tracking candidates by analyzing the evolution histories of a large set of clones might even take several months to complete. She feels helpless and realizes the necessity of a tool that can automatically analyze the clone evolution history and identify the important clones to be refactored or tracked. We believe that our tool SPCP-Miner can be her best friend in this situation. SPCP-Miner automatically mines and analyzes the evolution histories of the clone fragments and reports the most important clones to be refactored or tracked by ranking them according to the necessity of refactoring or tracking.

III. Concept behind the Tool

SPCP-Miner automatically mines the past evolution history of the clone fragments in a code-base and analyzes which clone fragments evolved by following a *Similarity Preserving Change Pattern* (*SPCP*). A *Similarity Preserving Change Pattern* consists of only *Similarity Preserving Change* and/or *Re-synchronizing Change*.

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied on this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered as clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. Let us consider two code fragments that are clones of each other in a particular revision. A commit operation was applied on this revision, and any one of both of the fragments received some changes in such a way that the code fragments were not considered as clones of each other (i.e., the code fragments *diverged*) in the next revision. However, in a later commit operation any one or both of the code fragments received some changes, and because of these changes the code fragments again became clones of each other (i.e., the code fragments *converged*). Such a *diverging change* followed by a *converging change* is termed as a re-synchronizing change. Fig. 1 shows a *Similarity Preserving*

Change Pattern containing *Similarity Preserving Change* and *Re-synchronizing Change*.

Separating the SPCP clones into two subsets. *SPCP-Miner* analyzes the evolutionary coupling of the SPCP clones, and separates these SPCP clones into two disjoint subsets: (1) *cross-boundary SPCP clones*, and (2) *non-cross-boundary SPCP clones* on the basis of this analysis. The SPCP clones in the first subset have evolutionary couplings (i.e., change couplings) with other code fragments (non-clone fragments or clone fragments from other clone classes) beyond their class boundaries. Thus, removal of such an SPCP clone fragment through refactoring might negatively affect the future evolution of the related code fragments beyond its class boundary [12]. Our empirical study [12] shows that *cross-boundary SPCP clones* are the most suitable ones for tracking. For the details on how we detect *cross-boundary SPCP clones* using *association rules* and constrained *support* and *confidence* values we refer the readers to our previous work [12].

Non-cross-boundary SPCP clones are the most suitable ones for refactoring. *SPCP-Miner* also makes groups of the non-cross-boundary SPCP clones. We conducted an in-depth empirical study [12] on the cross-boundary and non-cross-boundary SPCP clones. *SPCP-Miner* ranks the cross-boundary as well as non-cross-boundary SPCP clones on the basis of their change-proneness [12].

IV. Tool Description

Our tool [15] (SPCP-Miner) works on the output of a clone detector. Given the SVN repository URL of a subject system, our tool at first automatically extracts all the revisions of the system from the repository using *export* command of SVN, and then applies the clone detector to detect clones from each of the revisions. The user only specifies the SVN repository URL. The rest of the task is automatically done by our tool. After detecting clones from each of the extracted revisions *SPCP-Miner* performs eight sequential steps (c.f., Fig. 2) in order to detect the SPCP clone fragments. These steps are: (1) Method detection and extraction from each of the revisions using CTAGS[1], (2) Extraction of code clones for each revision from the clone detection results of the clone detector. (3) Detection of changes between every two consecutive revisions using *diff*, (4) Locating these changes to the already detected methods as well as clones of the corresponding revisions, (5) Locating the code clones detected from each revision to the methods of that revision, (6) Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [11], (7) Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy, and (8) Detection of SPCP clone fragments by analyzing clone change patterns. For the first seven steps we refer the interested readers to our earlier work [13].

Detection of SPCP Clone Fragments. Our tool considers clone fragments residing within methods. Before detecting SPCP clone fragments, SPCP-Miner detects method genealogies and clone genealogies considering all the revisions of the subject system. Detecting the genealogy for a particular method involves identifying each instance of that method in each of the revisions where the method was alive. By detecting the genealogy of a method, we can determine how it changed during evolution. We detect clone genealogies by locating the clones detected from each revision to the already detected methods of that revision. The genealogy of a particular clone fragment also helps us determine how it evolved through the commits. We assign unique IDs to the method genealogies and clone genealogies to recognize them across revisions. As we detect changes between revisions and reflect these changes to the methods as well as clones, we can examine how two clone fragments from a particular clone class changed during evolution by examining their genealogies. If these two clone fragments always received similarity preserving changes and/or re-synchronizing changes during evolution, then these are considered as a pair of SPCP clone fragments.

We determine all the SPCP clone pairs by examining all the clone genealogies. We merge these pairs to determine SPCP clone groups. If two different SPCP clone pairs have a common SPCP clone fragment, then we can say that the three clone fragments in these two pairs together followed a similarity preserving change pattern and thus, we can merge these pairs to make a group of three SPCP clone fragments. This group can also be merged with another pair or group if they share common SPCP clone fragments.

After detecting the SPCP clone groups, we analyze the evolutionary coupling of each of the SPCP clone fragments in each of these groups. If one or more SPCP clone fragments in a group have cross-boundary evolutionary couplings, then we consider this group for tracking. Otherwise, we consider the group for refactoring.

Implementation and Data Storage. We implement our tool using Java. Our tool stores data using files. While examining each revision of a subject system, it extracts the methods and clones in that revision and creates two separate files to store those. The changes between every two consecutive revisions are also stored in separate files. For a particular method we store the method name, signature, file path, starting and ending line numbers, class name (if any), and package name (if any). For each clone fragment we store the file path, starting and ending line numbers, and the clone class ID. After detecting the method and clone genealogies we provide unique IDs to the methods and clones. We update the files of methods and clones with these IDs. Here, we should note that we neither store the whole method body nor the actual clone fragment in the files. As we have just described we store enough information for a method or clone fragment so that we can get the corresponding code using the information.

Our experience during implementation. We would like to share our experience that we at first developed our tool using MySQL database for storing data. However, for subject systems with thousands of revisions each containing thousands of methods, the database becomes heavy-weight resulting in long response-times for complex queries. Then, we migrated from MySQL to simple file storage system and experienced that the implementation with file storage system is faster than the implementation with MySQL database by several folds. Currently *SPCP-Miner* supports four programming languages: C, Java, C#, and Python. However, we are working on it to support other languages too.

Improvements over Previous Implementation. We have already noted that we previously implemented *MARC* for the purpose of an empirical study [14]. *MARC* can also detect SPCP clones. However, it cannot separate the SPCP clones into *cross-boundary* (i.e., important for tracking) and *non-cross-boundary* (i.e., important for refactoring) groups. *SPCP-Miner*

[1]CTAGS: http://ctags.sourceforge.net/

can do this. Also, the SPCP clone detection strategy of *SPCP-Miner* is significantly improved as described below.

While detecting SPCP clone pairs *MARC* considers the following two constraints: **(1)** the two constituent SPCP clone fragments in a pair must co-change at least once during evolution, and **(2)** the two SPCP clone fragments in a pair must remain in two different methods. However, we implement *SPCP-Miner* such that it does not follow these constraints now. Firstly, according to the definition of similarity preserving change pattern (SPCP), two clone fragments from a particular clone class can follow an SPCP without being co-changed at all. So, we implement SPCP-Miner to detect all those SPCP clone pairs where the constituent clone fragments in a particular pair might not co-change at all. Secondly, two clone fragments remaining in the same method can also follow a similarity preserving change pattern and can be important candidates for refactoring. For this reason, we implement SPCP-Miner to consider the same method case too.

Tool Output. By working on all revisions of a subject system our tool generates: **(1)** An XML file containing the groups of all SPCP clones, **(2)** An XML file containing the groups of non-cross-boundary SPCP clones, and **(3)** An XML file containing the cross-boundary SPCP clones along with their coupled code fragments beyond their class boundaries. The groups of non-cross-boundary SPCP clones are important for refactoring. The cross-boundary SPCP clones are important for tracking along with the cross-boundary relationships.

Graphical User Interface. Fig. 3 shows a graphical user interface of SPCP-Miner. After running SPCP-Miner we specify the following things: the SVN repository URL of the subject system, the directory path to store the extracted revisions, the implementation language of the system, and the starting and ending revision numbers as shown in the figure. We save these information by clicking the button *Save System Information*. Then we click the button named *Extract Revisions* (at the left hand side of Fig. 3) to extract and store all the revisions of the subject system to the directory path that was input by the user. After the extractoin is done, we click the button named *Detect Clones from Revisions*. The tool then automatically detects clones from each of the extracted revisions by applying the clone detector. Here we should note that SPCP-Miner currently works with the NiCad [3] clone detector. NiCad helps us detect clones of major three types (Type 1, Type 2, Type 3) separately. Also, it is easy to extend SPCP-Miner for other clone detectors too.

At the left hand side of Fig. 3 there are five buttons in the section named *Preliminary Steps* and four buttons in the section *Detection of SPCP Clones*. The buttons in the *preliminary steps* are used for detecting and storing methods as well as method genealogies, extracting clones of different types (Type 1, Type 2, Type 3, or the combination of all types) from the clone detection results, detection of clone genealogies, and storing clone information along with clone genealogies. After completing these steps we can detect SPCP clones of different clone types using the buttons in the section *Detection of SPCP Clones*. If we click a button in this section, the tool will detect the SPCP clones of the corresponding clone type and store them in three XML files as we mentioned previously. The tool also shows all the SPCP clones along with the important ones for refactoring or tracking in three tables at the right hand side of the button panel as shown in the figure.

V. Limitations and Future Work

SPCP-Miner can detect cross-boundary SPCP clones which are more suitable for tracking rather than refactoring. However, SPCP-Miner does not yet support tracking of these clones. One interesting future work would be to build an Eclipse plug-in on top of SPCP-Miner to support tracking of such clones.

VI. Conclusion

In this paper we present a tool called *SPCP-Miner* which is capable of automatically identifying SPCP clones (i.e., the important clones from the perspectives of clone management) by examining the clone evolution history of a software system. SPCP-Miner also analyzes the evolutionary couplings of the SPCP clones and separates these SPCP clones into two disjoint sets. The clone fragments in one set are important for refactoring, and the clones in the other set are important for tracking. SPCP-Miner ranks the clones in these two sets on the basis of the necessity of refactoring and tracking. Our tool is the pioneer one in detecting important clones for refactoring and tracking. We believe that SPCP-Miner can complement the existing clone detection, refactoring, and tracking tools and thus, can help us in better management of code clones. SPCP-Miner is available on-line [15] with all necessary instructions for running it using an example subject system.

References

[1] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273 – 282.

[2] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, "A Novel Approach to Optimize Clone Refactoring Activity", Proc. *GECCO*, 2006, pp.1885-1892.

[3] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc *ICPC Tool Demo*, 2011, pp. 219 – 220.

[4] E. Duala-Ekoko, M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management", Proc. *ICSE*, 2008, pp. 843 – 846.

[5] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis", *Lecture Notes in Computer Science*, 2004, 3009: 220 – 233.

[6] Y. Higo, S. Kusumoto, "Identifying Duplicate Code Removal Opportunities Based on Co-Evolution Analysis", Proc. *IWPSE*, 2013, 5pp.

[7] P. Jablonski, D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE", Proc. *Eclipse Technology Exchange at OOPSLA*, 2007.

[8] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", *ESE* 13(6), 2008, pp. 645-692.

[9] M. Kim, V. Sazawal, D. Notkin, G. Murphy, "An empirical study on code clone genealogies", Proc. *FSE*, 2005, pp. 187 – 196.

[10] G. P. Krishnan, N. Tsantalis, "Unification and Refactoring of Clones", Proc. *CSMR-WCRE*, 2014, pp. 104 - 113.

[11] A. Lozano, and M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227-236.

[12] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking", Proc. *SCAM*, 2014, pp. 11 – 20.

[13] M. Mondal, C. K. Roy, and K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems", Proc. *CASCON*, 2012, pp. 205-219.

[14] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules", Proc. *CSMR-WCRE*, 2014, pp. 114 - 123.

[15] SPCPMiner. https://homepage.usask.ca/~mam815/spcpminer/index.php

[16] R. Tairas, and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities", *Information and Software Technology*, 2012, 54(12):1297 – 1307.

[17] M. Toomim, A. Begel, S. L. Graham. "Managing duplicated code with linked editing", Proc. *IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 173 – 180.

[18] M. F. Zibran, and C. K. Roy, "Conflict Aware Optimal Scheduling of Prioritised code clone refactoring", *IET Software*, 2013, pp. 167 – 186.

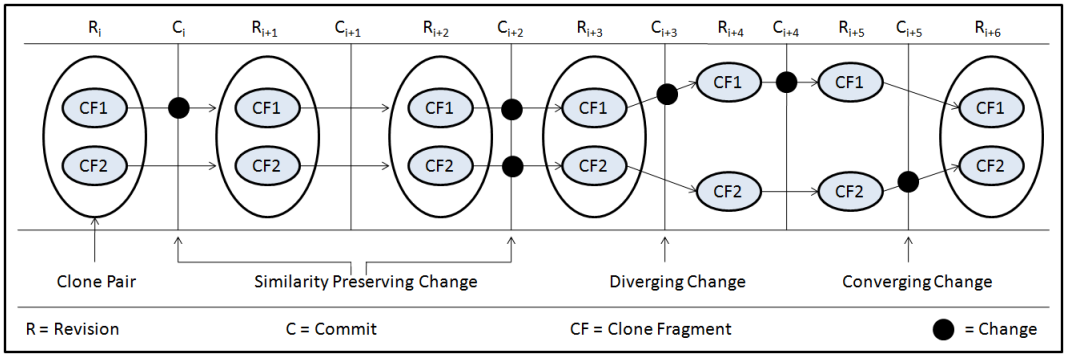

Fig. 1. A *Similarity Preserving Change Pattern* (SPCP) followed by two clone fragments *CF1* and *CF2*. We see that *CF1* and *CF2* received *Similarity Preserving Changes* in commits: C_i and C_{i+1}. They received diverging change at commit C_{i+3}. However, they again converged after the changes in commit C_{i+5}.

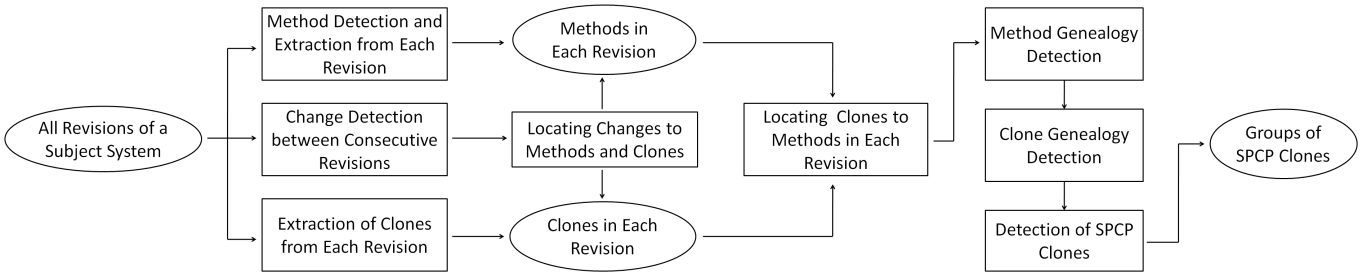

Fig. 2. The steps in detecting SPCP clones. There are eight processing steps in detecting SPCP clones. The rectangles in this figure indicate these steps.

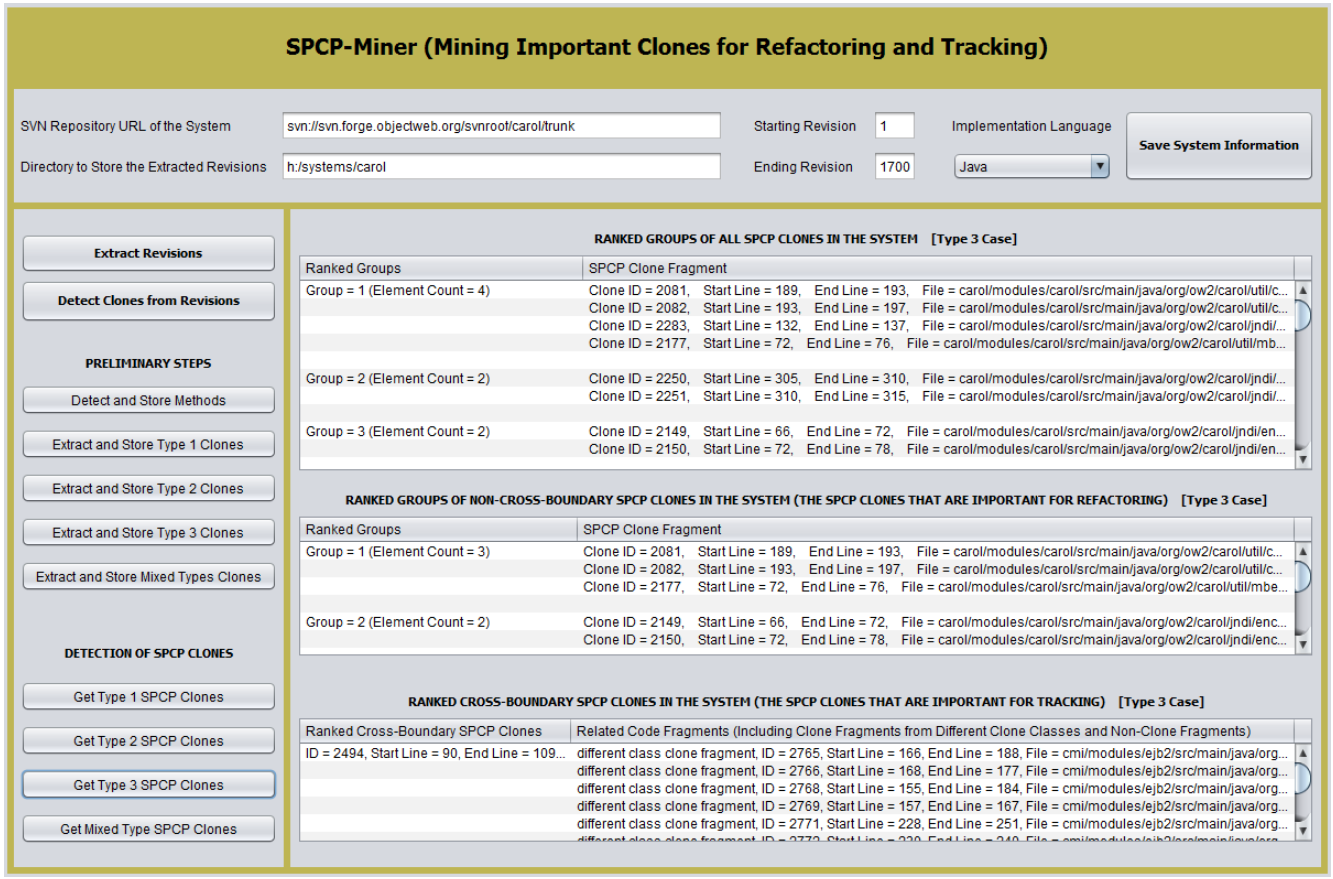

Fig. 3. The snap-shot of SPCP-Miner. In order to apply SPCP-Miner on a software system we input the system's SVN repository URL, the directory to store the revisions extracted from the SVN repository, the starting and ending revision numbers, and the implementation language. Then, we save these information by clicking the button *Save System Information*. We use the buttons at the left hand side to extract revisions, detect clones, and detect SPCP clones.