

Cleman: Comprehensive Clone Group Evolution Management

Tung Thanh Nguyen, Hoan A. Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{tung,hoan,nampham,jafar,tien}@iastate.edu

Abstract—Recent research results have shown more benefits of the management of code clones, rather than detecting and removing them. However, existing management approaches for code clone group evolution are still ad hoc, unsatisfactory, and limited. In this paper, we introduce a novel method for comprehensive code clone group management in evolving software. The core of our method is Cleman, an algorithmic framework that allows for a systematic construction of efficient and accurate clone group management tools. Clone group management is rigorously formulated by a formal model, which provides the foundation for Cleman framework. We use Cleman framework to build a clone group management tool that is able to detect high-quality clone groups and efficiently manage them when the software evolves. We also conduct an empirical evaluation on real-world systems to show the flexibility of Cleman framework and the efficiency, completeness, and incremental updatability of our tool.

I. INTRODUCTION

Code fragments that are exactly matched or similar to each other in a system are called *code clones*. The code clones with identical or similar logic require additional effort from developers in maintenance tasks. For example, they have to make sure that multiple clones are modified in a consistent manner. To deal with code clones, there have been several approaches including automatic *clone detection* [2], [4], *removal* of code clones [7], and *tracking* of clone evolution [5], [8]. However, recent research results have shown the benefits of the management of code clones, rather than detecting and removing them [5], [11], [12]. Some researchers concluded that the elimination of code clones could incur many trade-offs that in many situations may not be acceptable [11], [13].

Like other parts of source code, code clones themselves *evolve* as well during the software development and maintenance process. Many researchers have pointed out the need of automatic supports for clone evolution management [5], [6], [12]. However, there has never been any attempt to identify the key requirements for *automatic* clone management in evolving software. Let us propose the requirements for two most important tasks in clone group management:

Clone Grouping. Clone grouping mechanism is required for identifying clone groups and their members. Such mechanism should be *precise* (i.e. detected clone groups contain only true clones), *complete* (i.e. all cloned code are detected) and *concise* (i.e. the detected result is not redundant).

Clone Group Updating. When software changes, some clone groups might be created, deleted, or changed. Automatic

clone group management needs an *efficient* and *accurate* update mechanism of clone groups and their members. A good updating mechanism should enable the maintenance of the historical information on group evolution for developers to easily keep track of them over time.

In this paper, we introduce such a comprehensive method for clone management in evolving software. The heart of our method is *Cleman*, an general framework which relies on a formal model to characterize the important properties and those requirements for a comprehensive clone group management tool. Cleman allows one to systematically construct clone management tools satisfying the aforementioned requirements with few pre-defined criteria on cloned code. The framework is independent of the representation of code fragments and the criteria for the similarity measure between two cloned code fragments. Cleman is dedicated to the *management* aspect of clone groups and is flexible enough to support any types of code representation and code clone similarity measures.

To illustrate our framework, we have developed a prototype clone group management tool also called Cleman, which is able to identify and update the clone groups in an incremental manner during software evolution.

II. CLONE GROUP MANAGEMENT MODEL

In Cleman, a program is modeled as a collection of individual *fragments*. A fragment is a *portion of code of interests* in clone management such as a class, a method, or a code block. There are two relations defined among fragments. *Enclose relation* is to determine the structure of the fragments, i.e. whether one fragment is a part of another. The other relation, called *clone relation*, is to determine whether two fragments are clones of each other. It is symmetric but must not always be transitive. In addition, two different fragments in which one encloses the other cannot be considered as clones.

If two fragments are clones of each other, they are called a *clone pair*. A *clone group* is a set of at least two fragments of which any two fragments are clones. A clone group *covers* a clone pair if two members of the clone pair belong to the group, or are enclosed by some members of the group. A clone group *covers* another clone group if the former covers all clone pairs of the latter.

A *clone coverage* is a collection (i.e. a set) of clone groups such that any clone pair is covered by at least a group in that collection (i.e. the clone coverage “covers” all clone pairs).

A clone coverage is called *non-redundant* if there is no two groups in the coverage such that one group covers the other. A non-redundant clone coverage has no following redundancies: (1) a clone group contains both a fragment and one of its strictly enclosed fragments (because they cannot be considered as clones), and (2) a group is a subset of or is covered by another group.

A comprehensive clone management tool must produce and manage a non-redundant clone coverage. Among all non-redundant coverages, the ones that have smaller sizes are said to be more *concise* because they can cover all the clone pairs with a smaller number of groups.

Definition 1 (Clone Group Detection Problem): Given a program P as a collection of fragments, the corresponding enclose and clone relations, find a concise clone coverage C .

In software evolution, the program changes. The changes are modeled by a set of *inserted* fragments and a set of *deleted* fragments. The changes might affect the clone coverage, i.e. making changes to some groups. However, there might be groups that are absolutely unaffected and those groups must be kept in the coverage of the new program.

Definition 2 (Incremental Clone Update Problem): Given a program P , its current clone coverage C , and two change sets, *incremental clone updating* is to find a new concise clone coverage C' of the new program which contains all *unaffected* clone groups of the coverage C .

III. CLEMAN FRAMEWORK

Based on the aforementioned model, we present Cleman as a general framework that allows one to construct a class of solutions for the comprehensive clone grouping and updating. Especially, a solution in this class is able to efficiently produce and update a concise clone group result, i.e., a non-redundant clone coverage with a small size.

Note that efficiency is not captured in our formal model. This framework is only an approach for correctly implementing a solution for that model. The actual level of efficiency is dependent on the design choice and concrete implementation.

Cleman framework proposes three modules in a clone evolution management tool for clone grouping and updating:

- 1) **Fragment Builder:** This module is responsible for building the set of fragments and the enclose relation. It also extracts the necessary characteristic features of code fragments for the purpose of clone checking. When the program changes, Fragment Builder also detects the corresponding changes to the set of fragments, and construct the sets of inserted and deleted fragments.
- 2) **Clone Checker:** This module is in charge of checking whether two fragments are cloned or not. The precision of the tool depends on this module.
- 3) **Group Manager:** The responsibilities of this key module in Cleman include (1) clone grouping (for the first time run), and (2) clone updating (when the program changes).

Since there have been several well-studied solutions for the first two modules, Cleman framework focuses on providing the solution for clone group management.

A. Clone Grouping

The goal of clone grouping is to find a concise (non-redundant) clone coverage of a program. To improve its efficiency, Cleman employs a divide-and-conquer idea.

The first step is *mapping*. The code fragments are mapped, i.e. roughly filtered, into smaller sets called *buckets* such that the clone relation is *preserved* (i.e. any clone pair is mapped into at least one bucket). The efficiency of the following step (i.e. the step of finding the coverage for each bucket) is largely dependent on the size of a bucket. Therefore, each bucket should be as small as possible. To achieve this and still satisfy the requirement of clone relation preservation, a fragment might be mapped into multiple buckets.

The next step is to find a clone coverage R for each bucket. It starts with an empty coverage R and inserts fragments one by one. For each fragment v , the algorithm examines every group G in the current coverage and checks whether v can be inserted into G . v can only be inserted into G if and only if it is a clone of all fragments in G according to the clone relation. Otherwise, the algorithm creates a new group H containing v and all of its cloned fragments in G . This guarantees that all clone pairs of v with fragments in G are also covered by at least a group. H is then inserted into R .

Then, the coverages of all buckets are unioned to form a clone coverage C of the whole program. Because any clone pair is mapped into at least a bucket and then is contained in the coverage of that bucket, it is covered by C . Thus, C is consistent and complete (i.e. it contains all and only cloned code).

The relevant clone groups in C are combined into larger groups. The combination process can produce redundant groups. Therefore, at last, a filter step is applied to remove all redundant groups from the final coverage. This combination and filtering step makes the coverage more concise, but does not change the completeness and consistency of the coverage.

B. Clone Group Updating

When the program changes, clone group updating (i.e. computing new clone coverage C' for the new program) is performed in the following steps. Firstly, all groups in the current coverage C are copied into C' . Note that after grouping, the buckets are kept for clone group updating. Then, all deleted fragments are removed from the buckets and from the groups containing them in C' . All inserted fragments are re-mapped into the buckets. The removal and the re-mapping result in the changes to some buckets and might create new buckets. For each of those buckets, new clone coverage is recomputed (in the same way as in clone grouping). All new clone coverages of changed and new buckets are unioned to C' . Then, C' is combined and filtered in a similar manner as in clone grouping. Finally, it becomes a non-redundant coverage and all unaffected groups in the old coverage still remain.

IV. TOOL DEVELOPMENT

In Cleman, the design choices are available for Fragment Builder, Clone Checker, and the mapping module in clone

grouping (see section III-A). Those choices are interdependent. For example, the implementation of the mapping module will use the clone relation from Clone Checker, which in turn is dependent on the fragment representation and the enclose relation from Fragment Builder. Let us present our design choices and implementation of a clone group management tool also named *Cleman*. For experiment, we also implemented two alternatives: ClemanC and ClemanD. We built the Clone Group Manager and it was re-used in all implementations.

A. Cleman

1) *Fragment Builder*: In this implementation of Cleman tool, a fragment can be a class, a method, a code block or a statement, i.e. is a portion of source code having specific semantics, and is represented by a subtree of the program's Abstract Syntax Tree (AST). For each fragment, a *characteristic vector* and a *characteristic sequence* are calculated.

Characteristic vector is a vector of occurrences of relevant AST node types in the fragment. This kind of characteristic vector was used first by Deckard [9]. *Characteristic sequence* of each fragment is its sequence of nodes in the corresponding subtree which is traversed in the preorder. Note that this sequence contains the AST node types, rather than lexical tokens as in token-based clone detection tools [3].

For the enclose relation, we use the relation defined by the ancestor-descendant relationship among nodes of an AST. A fragment u is enclosed by another fragment v if the root of the corresponding subtree of u belongs to the corresponding subtree of v .

2) *Clone Checker*: In this implementation, two fragments u and v will be considered as clones if and only if

(a) They have same *type*. The type of a fragment is the type of the root node of its corresponding subtree. There are four types: class, method, block, and statement. Only fragments of the same type are considered cloned code;

(b) The Euclidean distance between two characteristic vectors is smaller than a user-specified threshold, and

(c) The likelihood ratio of their characteristic sequences is larger than a user-defined threshold. That likelihood ratio is defined as $2L/(L_1 + L_2)$ with L is the length of the largest common subsequence of the two characteristic sequences, and L_1, L_2 are their respective lengths.

3) *Mapping*: For the mapping module, our tool uses a hash scheme based on locality-sensitive hashing functions (LSH) [1]. Accordingly to [1], the probability that two vectors have the same hash code increases if their distance decreases. If we use a sufficiently large number of hash functions to map fragments into buckets based on their characteristic vectors, two cloned fragments that have the distance of their characteristic vectors smaller than a threshold will have a very high probability to be hashed into the same bucket. Moreover, our hashing scheme is implemented such that fragments of different types cannot have the same hash code (by using a part of the hash code to encode the fragment type). Therefore, only fragments of same type are hashed into the same bucket.

B. ClemanC

ClemanC is implemented using the same design choice as CCFinder [10] where fragments are represented as token sequences. Two fragments are cloned if two corresponding token sequences are identical. The mapping is a hash function for sequences. This hashing scheme is a complete mapping, because the cloned fragments that have the same token sequences will have the same hash code and will be mapped into the same bucket.

C. ClemanD

ClemanD uses the same design choices as Deckard [9] where each fragment corresponds to a subtree in an AST, and is represented as the vector of occurrences of AST node types. Two fragments are considered as clones if the distance of two corresponding vectors is smaller than a threshold. The mapping scheme is also based on LSH.

V. EMPIRICAL EVALUATION

Note that the clone group detected by a tool based on Cleman framework is always consistent (i.e. fully precise) with respect to the given clone relation. The actual precision of the tool is dependent on the degree of precision of the used Clone Checker module. In addition, the efficiency of the tool is not modeled in the framework and is dependent on the design and implementation of each module.

In general, there is a trade-off between completeness (i.e. recall) and efficiency. If one wants to achieve full completeness, he/she might use a pair-wise comparison to detect all clone pairs, and have each bucket containing only a clone pair. In our tool, we use multiple hashing. It might not achieve full completeness, but our experiment (will be described later) showed that the loss in completeness is relatively small compared with the gain in efficiency.

For the evaluation of Cleman, we conducted experimental studies with various real-world software systems. Our goal is to evaluate the flexibility, efficiency, precision, completeness, and incremental updatability of our method. For performance evaluation, we use the following measurements:

(a) *Precision* is defined as the percentage of the *correctly detected* clone groups in the *total detected* clone groups.

(b) *Cloned line coverage* is defined as the percentage of non-overlapping lines of detected clones in the total number of lines of code in a project. It was introduced in [10] and popularly used as an indicator of the *completeness* (i.e. *recall*) of clone detection for large projects.

(c) *Detection cost* is used to evaluate the efficiency of tools. It is defined as the ratio between the running time cost over the corresponding coverage value. This measurement reflects the efficiency better than the running time cost because it takes into consideration the level of completeness in the result.

A. Flexibility

The successful implementation of Cleman, ClemanC, and ClemanD shows that Cleman framework is flexible, i.e. different design choices for fragmentation, clone relation, and mapping can be used.

Three tools are run on a medium size project Apache Log4J (57kLOC). This project is small enough for us to manually check the precision of the results. The experimental result shows that ClemanC has very high precision but small cloned line coverage. It is because that ClemanC's clone relation accepts only the *exactly* matched fragments. ClemanD has the largest cloned line coverage but the lowest precision value. The reason is that the small vector distance is necessary but not sufficient for clone relation. Cleman has cloned line coverage much larger than ClemanC, comparable to ClemanD, but has much higher precision. It means that the design choices in Cleman are more reasonable than other two.

B. Efficiency and Completeness

In this experiment, we evaluate the efficiency along with the completeness of Cleman against two other popular tools SimScan and CCFinder [10]. For evaluating the trade-off between completeness and efficiency, we run Cleman with different hashing functions. Because chosen thresholds affect the coverage values, we use the same thresholds as the ones used in the experiment for flexibility, which have produced high precision values.

Our experimental result shows that Cleman is significantly faster and has lower detection costs than SimScan with comparable cloned line coverage values, and is slightly faster than CCFinder with much larger cloned line coverage values. It means that Cleman is more efficient than both CCFinder and SimScan.

The result also shows the trade-off between efficiency and completeness. Running on Axis, the largest project among subject systems, the result shows that using more hash functions results in better completeness, but less efficiency. However, the detection cost increases much faster than cloned line coverage. Therefore, the degree of losing in efficiency is far more than the degree of gaining in completeness.

C. Incremental Updatability

This experiment is to evaluate Cleman's ability of incremental updating. In this experiment, two consecutive versions of each project were used. Cleman tool was first run on the earlier version. Then, it was run in both incremental and independent modes for the later version. The result shows that the incremental and independent runs gave the same detection results (cloned line coverage values and clone groups). That is, Cleman is able to correctly update clone groups. However, the running time in the incremental mode (i.e. incremental updating) is much faster and the detection costs are lower. In addition, the less the code changes, the less detection cost is. Importantly, in the incremental mode, Cleman can keep track of the history of clone groups (changed and unchanged clone groups), and detect the removed clone groups and brand new cloned code.

For incremental updating, Cleman needs to store its internal data structure (e.g. fragments' information, buckets, and clone groups). Our experiment shows that the storage costs are

acceptable and the benefits of clone management and the gain in efficiency for updating far outweigh the storage costs.

VI. CONCLUSIONS

In this paper, we introduce Cleman, a novel approach to comprehensive management of code clone evolution. This approach formulates the requirements of comprehensive clone group management and provides a framework for systematically building such management tools satisfying the requirements. Based on our framework, a clone management tool is built and evaluated. The experiments on real-world systems show that our tool is able to efficiently detect higher quality clone groups than existing approaches, and fully update/track them in an incremental manner.

Our empirical evaluation also shows that our framework is flexible and can be used as an infrastructure for the exploration of the design space and the comparison of comprehensive clone group management tools.

ACKNOWLEDGEMENT

This project was funded in part by a grant from the Vietnam Education Foundation (VEF) for the first author. The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of VEF.

REFERENCES

- [1] A. Andoni and Piotr Indyk. E2LSH 0.1 User manual. <http://web.mit.edu/andoni/www/LSH/manual.pdf>.
- [2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *WCRE '99*, page 326–336. IEEE CS, 1999.
- [3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98*, page 368. IEEE CS, 1998.
- [4] R. Tairas - Bibliography of clone detection literature. <http://www.cis.uab.edu/tairasr/clones/literature/>.
- [5] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07*, pages 158–167. IEEE CS, 2007.
- [6] E. Duala-Ekoko and M. P. Robillard. CloneTracker: Tool support for code clone management. In *ICSE '08: Research Tool Demonstration*. ACM Press, 2008.
- [7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [8] P. Jablonski and D. Hou. CREN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX'07*, pages 16–20. ACM Press, 2007.
- [9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105. IEEE CS, 2007.
- [10] 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*, 28(7):654–670, 2002.
- [11] C. Kapsner and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE '04*, pages 85–94. IEEE CS, 2004.
- [12] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.
- [13] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE '07*, pages 116–126. IEEE CS, 2007.