Thomas LaToza
5/5/2005

A Literature Review of Clone Detection Analysis

Introduction

Code clones, pieces of code similar enough to be considered duplicates or clones of the same functionality, are a problem. Despite beliefs that code should never be copied and pasted and agile dictums that all duplication be removed, there are frequently good reasons to copy and paste code. Developers wish to do be able to do something implemented elsewhere that they can't call directly. This might be because the code makes assumptions or design decisions that the developers need to change. Or the code might be contained in a module that can't be called either for architectural reasons or even for organizational reasons. Clones may also be reimplemented code where developers repeatedly introduce the same code into the code base. Developers working on large code bases might not know that someone has already solved their problem, especially in cases where the solution is treated as a secret hidden behind interfaces rather than exposed functionality to be reused. Developers may reimplement quicksort or a graph traversal algorithm by copying code from google or looking it up in their favorite textbook.

More generally, creating abstractions to remove duplication represents a significant investment that may not always be immediately justified (Kim and Notkin, 2004). In situations in which modularity has failed and prevented developers from doing what they need to do without introducing duplication, refactoring may require large amounts of knowledge about the architecture and design. Moreover, in some situations, developers do not know how fast or how far the code they have copied will diverge as their design is uncertain or future requirements to implement are uncertain. Creating an abstraction is thus also a risk. If the abstraction is not created, the clones might soon diverge so far that they are no longer clones. Introducing the abstraction then makes the code potentially more complicated in having to support widely divergent uses. Developers forced to choose an abstraction to introduce right when copying and pasting may not have enough information to choose the correct abstraction. When the abstraction must subsequently be changed, this introduces large amounts of rework. Or if the abstraction is not changed, the code may be more difficult to read and more tedious and time consuming to implement.

Code duplication may also happen for performance reasons. In limited memory environments, it may be necessary to have specialized versions of the code that do not contain extraneous instructions or variables that use valuable memory. This may lead to multiple implementations that are mostly identical.

For all of these reasons, code clones are a significant problem. Empirical assessments of the number of clones in real systems have typically yielded numbers ranging from 15-20% of LOC. For instance, CCFinder, a mature code clone analysis, reports that 21.35%

of JDK is cloned code (Kamiya et al, 2002), although the authors note that some of this is attributable to generated code.

Clone detection tools

One important way to fight code clones is to identify existing code clones in the source. Developers can then evaluate whether removing the clones by refactoring to introduce an abstraction is justified. Developers also have less reason to prematurely introduce an abstraction as the difficulty of introducing an abstraction later is reduced. Clone detection tools provide an automated means for developers to find and inspect code clones.

Clone detection tool authors argue that they will never provide a completely automated tool as whether or not code should be considered a clone requires developer judgment. The amount of code in common between clones, the number of clones of the same code, and the difficulty of refactoring all determine how the developer should act when confronted with a potential clone. Thus, current tools are built with the idea of trying to help developers explore the clones rather than simply providing a list of clones to remove or trying to automatically suggest refactorings that the developers should take. The tool authors do not believe that tools ever should be making engineering decisions about how aggressively developers should remove clones.

A typical clone inspection GUI consists of a matrix. On each axis of the matrix, a list of tokens (to be defined later) is shown. Each grid cell is marked if the two tokens in common are considered by the tool to be a clone. The grid allows developers to inspect a sequence of common tokens which together constitute a clone. For instance, a line of code tool might report that 10 lines of code are clone lines by showing that 10 contiguous cells in the grid are clones. This lets the developer see the size of the clone and see if any other clones are next to it and might also be logically part of the same code fragment, even if they are separated by several nonclone tokens. The tool may also link to the source code and allow clicking on clone entities to navigate the source code editor.

Most current clone tools presume some sort of batch mode where the tool is run for a number of minutes on the codebase, results are produced, and the developer investigates the results. Current tools do not support persisting this information and incrementally updating it to allow the user to have this information available for actual minute to minute programming tasks. A clone detection tool is something that needs to be started and used separately.

Some more sophisticated tools provide metrics about clones to make it easier for developer to choose which clones they wish to investigate the source of and which clones are most worth refactoring. CCFinder provides information about clone classes, or sets of clones. It provides a maximum length of any clone in the class in either LOC or programming language tokens. It provides the number of clones in the clone class. It provides a deflation index which estimates the net LOC that would be removed by replacing all of the instances of the clone with a common function and adding call statements to the new function to all of the clone instances. Radius estimates the

difficulty of maintaining the code clones consistently by measuring the maximum path over all clones up the directory tree and back down the directory tree to the furthest away clone. This is used as an approximation of organizational distance of the two clones; clones that are further away thus have a higher chance of being owned by different developers or organizations, making it more difficult to change both clones.

Code clone tools are similar to other string matching tools used for other software engineering tasks. Code clone tools are similar to diff tools used by version control systems to find the maximum similarity between an old version of a file and a new version of a file. Both rely on string matching algorithms. Yet, clone tools are different in that they look for multiple copies of the same piece of code instead of just finding the minimal transformation from one version to the next. Even a single file might contain multiple copies of the same code clone.

Besides being used to detect code duplication, clone detection algorithms and tools have also found other uses. CCFinder was used to attempt to determine whether a company stole source code. One company filed suit against another for stealing its code and submitted evidence based on CCFinder. CCFinder determined that over 50% of the files contained clones that were also in the other company's source code.

Clone detection tools are also similar to plagiarism detectors used to detect student cheating. MOSS (Schleimer et al, 2003) is a generic document comparison tool used to ascertain whether too documents have been copied from each other. Its web based front end for source code has been widely used by computer science instructors to process student submissions to detect plagiarism. It also faces similar issues of having to filter out uninteresting copying in that some copying is sanctioned by the instructor. The instructor may hand out given code or have a template for header or documentation information. MOSS thus lets the user declare that certain information is ineligible to be considered a duplicate.

Desiderata for clone detection algorithms

A clone detection algorithm is what allows a clone detection tool to determine which pieces of code are clones. Some tools, like CCFinder, support detecting not only pairwise clones between code pieces, but also compress pairs into clone equivalence classes by defining a clone equivalence relation. A clone equivalence relation is a reflexive, symmetric, transitive relation over two clones that describes a class of clones in which every pair of clones in the class satisfies the relation.

A clone detection algorithm must possess the ability to exclude uninteresting clones. Statements such as printf("\n") are likely to be frequent clones in many codebases. Most code has statements that occur frequently. Yet, 300 copies of printf("\n") in different places likely does not represent duplication the developer cares about. In most cases, these statements wouldn't be worthwhile extracting into their own shared method. Yet, if this represents a secret that should be hid, this might then be considered an interesting clone that should be flagged. Thus, algorithms need to be configurable so that developers can select what constitutes an interesting code clone for the system, coding conventions,

and task at hand. The tool will not be useful if a developer must sift through many false positives that are not useful.

A clone detection algorithm must also be able to understand clone subsumption to find larger clones rather than smaller clones. Small strings may match between two pieces of code. Yet, if these strings are part of larger strings that are also clones, the strings that are contained by the larger strings should no longer be considered clones in their own right. Clone detection algorithms are thus also interested in finding the minimal number of clones that still captures all of the individual duplicated tokens in a codebase.

Clone detection algorithms also need to be resilient to incidental change. Once a piece of code has been copied and pasted in its new location, it will likely be at least superficially modified in having some variables renamed, statements reordered, statements added, or calls changed. Rather than simply finding only perfect clones that are exact matches, a clone detection algorithm needs to be resilient to these incidental changes up to a point and still classify code as being a clone.

Finally, like any program analysis, clone detection algorithms need to be fast enough that they can be run on real programs. Some clone detection tools try to skirt this issue by having the user select only some subset of the code in which to look for clones. But, ideally, a clone detection algorithm should be able to run in a reasonable amount of time on a multimillion line codebase.

Clone detection algorithm

Clone detection algorithms can be simply described as particular design decisions parameterizing a single common algorithm:

```
DetectClones (source)
        Tokenize the source
        Transform tokens into canonical form
        Match tokens to generate candidate clones
                If candidate clone passes filter, mark as clone
```

DetectClones first parses the source code into some token representation. The tokens are then transformed by throwing away some information to create a canonical abstract representation. The canonical abstract representation then allows the matching process to compare tokens to determine whether they are the same canonical token. If the candidates pass both the matching process and an optional filtering process used to remove candidates that are clones but aren't actually interesting, the match is marked as a clone.

The matching step is the slowest and generally dominates the runtime. Most matching algorithms do some sort of string matching to be able to detect the clones as longest common subsequences in the token stream.

Clone detection algorithms are best distinguished by their definition of a token. The oldest and simplest is to simply consider every line of code as a token. This is fast but misses clones that differ in many types of incidental changes. The next most powerful is to use programming language level tokens. CCFinder, a tool based on this technique, is the most mature copy and paste detector. Even more resilient to incidental changes, AST nodes allow reordered clones to still be detected by making it easier to build a hash that throws away this ordering information. However, they are also slower. The slowest copy and paste detector is based on slicing. Computing slices is expensive, but this algorithm is forced to do it repeatedly to detect and grow matching tokens into the maximal clone. Thus, the implementation is really slow. Origin analysis (Godfry and Zou, 2005) attempts to determine where a piece of code originated from by tracking its movement from checkin to checkin. It uses callgraph nodes as tokens and compares between versions of source code rather than within a particular version of the source code.

Line of code clone analysis

The oldest and simplest level of granularity to consider for clone analysis is a line of code. This approach is implemented in the tool Dup (Baker, 1995). Parsing into tokens is trivial. The tool implements a number of transformations once the program is in token form. Whitespace and comments are removed, preventing extra spaces or comments inserted in the middle of code copied and pasted from disrupting a match. More importantly, identifiers are stripped and replaced with metavariables. This allows Dup to detect cases where the developer has changed a variable name to some locally appropriate value. Yet, the use of line of code granularity means that Dup will miss any sort of line breaks. Each piece of the line will end up as a separate token, and the inserted token will cause string matching to fail.

Dup only implements a single type of filter – number of lines of the clone. This allows it to exclude simple things like braces or print("\n"); statements as the tool produces way too many matches to be useful at levels of only a few lines. Dup also presents statistics of the percentage of clones in different modules, the number of clones, and the reduction in number of statements from removing clones. These help the developer make better decisions on which clones are worth refactoring.

Dup uses a suffix tree string matching algorithm to match transformed line tokens against each other. Thus, Dup's overall running time is O(|LOC|). It is able to process 1 MLOC in 7 minutes on a 40 MHz processor.

Language token clone analysis

The current most effective balance between speed, safety, and soundness is to use a programming language token based approach. CCFinder (Kamiya et al, 2002) uses this form of the clone detection algorithm. Parsing into tokens is relatively easy in that a standard programming language lexer can be used. CCFinder's real sophistication comes in the form of a number of sophisticated rules for transforming and filtering tokens.

A number of tokens are added, removed, and modified during the transformation process. CCFinder attempts to easily support analysis of multiple languages, so there are separate sets of transformation rules for C, Cobol, and Java. These rules describe a number of types of incidental changes common to the programming language and give rules by which the tokens can be translated to remove the incidental changes. For instance, the Java rules strip package names and readd class names when a function is only implicitly using this in a member variable reference or member function call. Initialization lists are stripped, accessibility keywords are stripped, and choice constructs are all forced to have a compound block inside of them. These transformations help to increase the tool's effectiveness by minimizing the number of incidental changes that will cause a clone to be missed.

A number of filtering rules are applied later before a clone candidate can be presented as a clone. Clones must begin at the start of a basic block. Several types of repeat clones (such as case statements and variable declarations) are disregarded.

CCFinder uses a standard suffix tree string matching algorithm. It's runtime is O(LENGTH(longest clone) * |Tokens|). Since the length of the longest clone can generally be considered as a constant, this reduces the runtime to simply O(|Tokens|). The use of a simple analysis for obtaining and matching the tokens makes CCFinder run fast. It completes 10 MLOC in only 68 minutes on a 650MHz processor.

The developers claim a number of unique benefits for CCFinder. They point out that by using transformation rules rather than a more sophisticated program analysis, they are able to minimize the dependence on their tool to these simple, declarative transformation rules that are easy to build for a new language. They have already made these transformation rule libraries for Java, C, and Cobol. They also note that they implement finding clone classes rather than simply code clones. This allows developers to quickly see that there are many copies of the same clone rather than having to do the transitive closure by hand by looking at all of the pairwise relationships.

Abstract syntax tree clone analysis

Abstract syntax tree clone analysis (Baxter et al, 1998) attempts to be more accurate than a line or programming language token based approach by building the abstract syntax tree. Before taking advantage of its AST representation, the tool first expands macros to ensure that all of the information will be in the AST. After building the AST, a hash of each of the AST subtrees is performed. This removes identifiers. Comments and white space have already been removed by building the AST node tokens.

Matching relies on hashes for each of the AST subtrees in the AST. The step first places all subsequences of the same length in similar buckets based on the similarity of the hash. All of the subsequences in this bucket are then compared against the similarity threshold. The subsequences that pass are then passed on to a generalization process that visits the parents of the clone AST nodes until a set of parents is found that is not a code clone. Thus, the algorithm requires that clones match by exceeding the similarity threshold at each particular AST node in the hierarchy.

The algorithm's reliance on building all of the AST subtrees and doing relatively pricey AST operations makes it significantly slower than most other tools. The algorithm itself is O(|Subtrees of AST|). It runs in 120 minutes for 100 KLOC. The more powerful AST manipulations and AST approach allows the tool to correctly detect statement reorders and statement insertions.

Slice based clone analysis

Even more powerful than merely an AST approach, a tool (Komondoor and Horwitz, 2001) has also used every node in a program dependence graph to form slices to compare. A program dependence graph adds edges between statements whenever a data value depends on another statement for its value. These edges are either control or data dependencies. The token creation step consists of simply creating the program dependence graph from the source.

Next, all of the program dependence graph nodes are partitioned into equivalence classes based on syntactic similarity of the statements. Differences in identifier or literal values are ignored. For each initial pair of program dependence nodes in the equivalence class, generalization proceeds to find the largest isomorphic subgraphs of the program dependence graph including the two initial nodes. Backward and forward slices are added to increase the size of the isomorphic subgraph until it is not possible to add any more slices.

Unfortunately, the use of slicing makes the algorithm very slow. The tool takes 13 minutes to run on just 3419 LOC! The use of slicing does make it the most accurate algorithm, however. Unlike the AST approach, similar clones do not have to include all children of some AST node parent. The approach can now detect fully entangled clones where the actual cloned lines of code are spread far apart and only linked by program dependence graph dependencies. It could thus be used to handle older or much more heavily modified clones where the original copy and paste code has been spread apart by the insertion and refactoring of functionality. It also easily handles statement reorders.

Call graph node origin analysis

An approach very similar to clone detection analysis has been used to track the movement of code over time in a process called origin analysis (Godfry and Zou, 2005). Origin analysis attempts to ascertain, for every function, the function that it came from in the previous version. This is interesting when functions are split, merged, and renamed, as a simple name search through the previous version will not establish a linkage with the current version. Origin analysis operates on the call graph, where every call graph node is a token. It attempts to match a function in one version with the most similar function in a previous version.

Rather than having a single transformation for each token, origin analysis provides a variety of "matchers" that both transform the call graph node and rate similarity between clone candidates. Overall similarity can be any combination of the individual matchers. The name matcher finds the longest common substring of two functions. The metrics

matcher calculates the weighted sum of LOC, fan in/out, # variables, and cyclomatic complexity. The declaration matcher finds the longest common substring of the lexically sorted parameter identifiers. Finally the call relation matcher finds the size of the intersection between candidates' caller and callee functions. It is most useful for looking at splitting, merging, and renaming.

Rather than run all of the matchers over all of the code in batch mode, the origin analysis tool provides an incremental, as needed analysis. The user must select a set of candidate functions in each version, and the tool will then attempt to identify the origins of particular functions. This analysis is then "almost instantaneous" rather than taking a long time. Moreover, the user may wish to switch matchers, matcher parameters, or matcher weights. Being interactive allows the developer to quickly try all of these options without having to wait for a long batch computation to complete.

Comparison of Algorithms

The algorithms all strike compromises between providing more accurate results and running in a usable amount of time. To get better results, there are two main approaches – use a more powerful analysis (AST or slicing) or build heuristics that filter specific cases of unwanted clones (CCFinder, Dup).

Towards building better heuristics, CCFinder's creators have spent time evaluating transformation rules on real systems including looking at JDK, Linux, NetBSD, and FreeBSD. This allows them to evaluate their transformation rules and understand what rules are necessary to work well on real systems. The improvements in their accuracy and performance will likely come through this empirical study of exactly how clones change rather than solving hard algorithms problem.

In contrast, AST and Slicing approaches are much more algorithm centric in relying on better program analysis to do their work for them. For them, the challenges that lie ahead will be coming up with faster algorithms.

Research Directions

Despite a number of tools over the course of a decade and several recent empirical studies of code clones, there is still no solid definition of what constitutes a clone. This is because clones inherently carry with them some values about engineering tradeoffs – are these duplicated pieces of code potentially worthwhile to factor out and remove. A better definition would allow the creation of benchmarks with which to make more meaningful comparisons of an approach's ability to find not just classes of clones but individual clones. Yet today, even people can't decide on what does or does not constitute a clone (Walenstein et al, 2003). A definition of code clones will probably need to entail some description and understanding of when code clones are significant enough to potentially be worth refactoring. This definition needs to be sufficiently formal that the clone detection tools can use it to filter clone candidates.

While related, there is also a need for a better understanding of what developers are likely to change right after performing a copy and paste. Attempts to build heuristics will need

to consider not just what is easy to detect but what types of changes developers actually make. This type of information could be relatively easily gathered by simply logging everything a developer is doing and examining the data around the time of copies and pastes. Having this data would lead to better benchmarks of what needs to be prioritized.

One area that none of the tools other than perhaps slicing has gotten close to is being able to detect reimplementation clones. Syntactic or likely even AST based techniques rely on information that is probably too low level to catch clones that have as a common source only being copied from some abstract algorithm in a textbook. On the other hand, because the tools being used to count duplication do not detect this, it is not even clear that this is a significant problem worth solving. Some empirical study trying to examine what types of reimplementation clones exist in a system, how frequent they are, and how they could be best detected would probably be the most useful way to proceed.

Clones also have the potential for a tie in with the movement towards making recipes and protocols more explicit in the design. A code clone could be considered as a type of recipe for solving some problem. Attempts to document common recipes for solving important tasks might be extended to encompass any task that it repetitive enough to require a developer to use copy and paste to implement it. Linking together the clones to the recipe would also remove much of their harm to the changeability of code.

Existing tools seem to presume a reengineering or a perfective maintenance scenario in generally being batch oriented. Developers are forced to wade through a separate matrix of potential clones and have to launch and wait for the tool just to see any results. Seeing clone links on the left eclipse editor bar would probably be a much nicer interaction in allowing developers to be reminded of clones when they are working with the code in question. This allows them to immediately take the clone information into account when beginning to consider any change to one or the possibility of building a new abstraction.

Finally, one simple way of much more reliably detecting copy and paste code clones would simply be to log copy and paste. Each clone could receive some XML comment or annotation that contains a unique identifier for the clone class or a listing of links to other instances of the clone. Such a solution would not be as valuable for a really messy legacy system but would help contain the problem of clones.

Conclusions

Code clones are an important problem. None of the existing tools have yet to have any impact on the problem in the real world – none of the tools have yet reached widespread penetration. This is partially because CCFinder is one of the few tools that does a good job providing only interesting clones while still running in a reasonable amount of time. CCFinder has already been used for an empirical software engineering study looking at the frequency with which code clones diverge after their creation (Kim and Notkin, 2004). But it is not open to download without the permission of its developers. As other tools mature towards being even more usable, the incentives for not sharing openly will diminish and the race to gain users may begin. In several years, code clone plugins may become a standard part of Eclipse.

References

Baker, B. On Finding Duplication and Near-Duplication in Large Software Systems. In *Working Conference on Reverse Engineering*, 1995, 86-95.

Baxter, I., Yahin, A, Moura, L, Sant' Anna, M., and Bier, L. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, 1998.

Godfry, M., and Zou, L. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. In *IEEE Transactions on Software Engineering*, 31, 2 (Feb. 2005), 166-181.

Kamiya, T., Kusomoto, S., and Inoue, K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28, 7 (July 2002), 654-670.

Kim, M., and Notkin, D. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In *Workshop on Mining Software Repositories*, 2005.

Komondoor, R., and Horwitz, S. Using Slicing to Identify Duplication in Source Code. In *Static Analysis Symposium*, 2001.

Schleimer, S., Wilkerson, D., and Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of SigMod*, 2003.

Walenstein, A., Jyoti, N., Li, J., Yang, Y., and Lakhotia, A. Problems Creating Task-relevant Clone Detection Reference Data. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, 2003.