2010 17th Working Conference on Reverse Engineering

From Whence It Came:
Detecting Source Code Clones by Analyzing Assembler

Ian J. Davis and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{ijdavis, migod}@uwaterloo.ca

Abstract

To date, most clone detection techniques have concentrated on various forms of source code analysis, often by analyzing token streams. In this paper, we introduce a complementary technique of analyzing generated assembler for clones. This approach is appealing as it is mostly impervious to trivial changes in the source, with compilation serving as a kind of normalization technique. We have built detectors to analyze both Java VM code as well as GCC Linux assembler for C and C++. In the paper, we describe our approach and show how it can serve as a valuable complementary semantic approach to syntactic source code based detection.

Keywords: Clone Detection, Reverse Engineering

1. Introduction

Source code clone detection is the attempt to discover similar logic within different parts of a code base, either so that cloned source code might be consolidated, making the resulting code cleaner [4], or so that the strong similarities between code in diverse parts of the code base might at least be recognized and documented to aid future maintainers. Clone detection has several potential benefits: it can ease software maintenance; the correcting of errors in all similar places where they occur may be more likely; and when performed on assembler it can aid in reverse engineering large systems for which there is no source code. In addition clone detection may serve as a possible metric of cohesion when evaluating unfamiliar code [3] [6] [7].

Two immediate challenges of any tool that performs source code clone detection are to discard non-significant differences in the source code, and to discover commonality within the logic that remains. These challenges can be addressed by developing a sophisticated clone detection tool that is capable of understanding all the intricacies of specific programming languages, or by having the clone detection tool operate on an intermediate representation of the source code [9] in which irrelevant differences in the source code are removed through normalization and complex control flow patterns are reduced to simple sequences of operations.

We have taken the latter approach for several reasons: clone detection can be cleanly divided into two independent sub-problems, simplifying the process; it permits the clone detection tool to operate on simplified input; and source code written in different languages — such as C, C++, and C# — can be treated in a homogenous rather than a heterogeneous manner. While it has been suggested [10] that this intermediate representation might itself be chosen so as to be of most use to achieving clone detection, there is some benefit in exploiting the intermediary assembler (or interpretive language) that is produced when source is compiled [2] [8].

This approach has the disadvantage that we must have access to existing binaries, or must be able to compile the source code. Clone detection tools that use token-based approaches do not make this requirement. Our approach also has the disadvantage that some compilers (notably not Java) generate assembler languages that are platform specific, while different compilers for the same machine might produce very different assembler.

However, our approach has the advantage that the input operated upon by the clone detector is known to be a genuine representation of the source code as used in the actual build process, accurately reflecting files included, macros employed, data types used, etc. This input can readily be derived from source code written in various languages without any need to develop, support and maintain custom translation tools for each such language. Many syntactic differences in the source are normalized in the assembler. A further advantage of this approach is that it permits clone detection to also be performed on object code for which source code is lacking. It

1095-1350/10 $26.00 © 2010 IEEE
DOI 10.1109/WCRE.2010.35

also cleanly permits clone detection to be performed on source code that contains embedded assembler. Finally, it is believed that semantic analysis of assembler permits discovery of logically equivalent code more easily than would be possible at the source code level.

The rest of the paper is structured as follows: In Sections 2 and 3, we discuss our approaches for analyzing assembler produced from Java and C/C++ source respectively. In Sections 4, we discuss the general approach that we take in detecting clones in assembly languages. Finally, in Section 5, we summarize our contributions.

2. Java

Java source code is typically compiled into one or more Java class files. It is these class files that the Java Virtual Machine reads when interpretively executing a program. These class files contain all of the semantic knowledge about what an arbitrary Java program is to do when executed. In addition, when Java programs are compiled in a manner that permits them to be later debugged, considerable extra information is embedded within the Java class files. This information permits correlation between the contents of these class files and the source code from which they are produced, as well as preservation of syntactic information such as the names of variables used in the source code.

The Java virtual machine specification provides extensive documentation on the internal format and interpretation of such Java class files. Using this standard it is straightforward to reverse engineer and mine the contents of Java class files so as to enhance the knowledge about the source code from which it is produced. Armed with this ability to correctly parse Java class files, we have developed software to reconstruct quasi-source code from a class file, to extract graphs that can be viewed visually showing a wide variety of different types of relationships which arise in Java source code, and to perform clone detection on Java.

The underlying common framework for parsing Java class files permits class files to be automatically constructed before being read by transparently compiling the corresponding source files; furthermore, it permits dynamic inclusion of new class files in the set to be examined, when referenced either directly or indirectly by the initial sources presented to clone detection. This inclusion uses the same class path mechanism that the Java compiler uses to discover needed components when compiling Java source code. Thus our Java clone detector can detect code duplication not only in the source code presented to it, but across the totality of all Java used in the execution of a Java program.

To facilitate better clone detection, we perform a small amount of post processing on the Java byte-code. Compression techniques that used different p-code instructions to describe the same operation are normalized. If local variable names are available (as consequence of the class files having been compiled using the –g option) these can replace internal variable numbers. Pseudo instructions are also added to reflect the start and end of try and catch blocks, since this information is not included within the p-code, but maintained in separate tables.

3. C, C++, and Assembler

The task of converting C and C++ code to assembler is automated by capturing a history of all of the compilation steps performed in the process of building a system, and then modifying the compilation parameters so that the desired assembler (when absent) can be automatically generated from each source file involved in that build process.

Each assembler file is read into memory, and then modified to better suit the needs of clone detection. The assembler is normalized by removing comments and redundant white space, and jumps are resolved to the assembler instruction branched to. References to constant strings are replaced by the strings themselves to simplify comparison of code that used such references. References to variables (within the assembler expressed as machine addresses) are replaced by the corresponding variable name, using the symbolic debugging information embedded within the assembler. Operations performed on parts of a variable (as for example happens when operating on double precision numbers or complex structures) are represented as variable name together with numeric byte offset within it.

Compiler-generated temporary variables also need names. Making the presumption that compiler generated names have scope restricted to the translation of individual lines of source code, we give such variables incremental numbers (in the order encountered), with the incrementing mechanism being reset each time the assembler being examined pertains to a new corresponding source line.

It remains unclear whether nested components of composite structures are best named as they would be within C and C++, which would cause clone detection to treat such names differently when any part of them changed, or as offsets from the root variable name, which would cause clone detection to treat them differently if internal memory addresses of such subcomponents changed as result of modification of the internal descriptions of the structures containing them. Possibly variables need matching against multiple naming schemes.

4. Steps in performing clone detection

The task of clone detection at the assembly language level is to discover maximal pairings of distinct matched assembler instructions from two distinct assembler subsequences (contained within a function) that occur in the same sequence, subject to some limit being imposed on the interleaved assembler instructions that are discovered not to match. We take a search-based approach [5] to performing the matching.

We begin by reading each class/assembler file into an *assembler* object. Assembler objects form a linked list which may be traversed. Each assembler object contained an array of *function* objects. Function objects themselves contain an array of p-code/assembler *instructions* which clone detection operated upon. The unit of comparison is typically a single assembler instruction. However, switch and case statements that are implemented as a lookup and branch table in the assembler are treated as composite instructions involving an ordered sequence of branch instructions by the clone detection tool.

Each instruction is stored in a hash table that with high probability will hash instructions deemed not to match when compared for equivalence to different hash chains. Each instruction records a back reference to the function containing it, and each function likewise contains a back reference to the assembler or class file containing it. The source and line number from which each instruction is produced is also captured, if available.

Our clone detection algorithm examines each instruction by walking through each function and each array of instructions within a function. An initial instruction P_1 may be deemed capable of matching some initial instruction Q_1 only if both are the first instruction generated at a given line in the corresponding source code, neither is derived from source in a location other than the primary source file from which the assembler derives (i.e. a macro defined within a header file) and/or only if both instructions do not require information from the stack added by an earlier instruction. These optional restrictions are designed to avoid matching sequences of instructions that are deemed not to be strongly related to the actual source code from which the assembler is derived.

To avoid the reporting of clones that overlap other clones, reported pairs of matched instructions may not subsequently be treated as the start of any other later clone.

Each matchable instruction is compared to all later instructions within the same hash chain (thus having the same hash value). For each pair of instructions that match, two sub-arrays of instructions are submitted to clone detection. These start at each matched instruction and typically run to the last instruction in the function containing it. However, when both instructions occur within the same function, the earlier sub-array has as its last member the instruction before the later matching instruction.

For most instructions determining if another instruction matches it is straightforward: it matches if and only if the instruction type matches, as well as any arguments associated with these instructions. Branch instructions, whose argument is an address, require more complex analysis. Self loops match only self loops. Branches outside the range of the instructions that potentially form a clone do not match. Otherwise two branch instructions match if they are of the same type, and either both branch forwards or both branch backwards and the nearest recognized matched instructions at or prior to the addresses branched to match one with the other.

Having identified an initial match, the length of the corresponding clone is determined as follows. A positive weight (default 1) is associated with cases where instructions match, and a negative weight (default -1) is associated with cases where instructions fail to match. More complex schemes might weight different types of assembler instruction differently.

A greedy (not necessarily optimal) algorithm is used that attempts to identify the longest subsequences beginning at the initially matched instructions having the property that the weighted sum of matched and mismatched instructions remains non-negative.

Given two sequences of comparable instructions P and Q, (by construction) P_1 matches Q_1. So suppose that we have just successfully matched instructions P_i and Q_j If either is the last instruction in one of the subsequences presented to clone detection we are done. Otherwise we attempt to extend the matched sequence by comparing P_{i+1} with Q_{j+1}. If these instructions also match we increase our weight associated with the matched subsequences and repeat. Otherwise we conclude that there must be at least one mismatched instruction, decrement our weight, terminate if it becomes negative and compare (subject to terms existing) P_{i+1} with $Q_{j+2,}$ and if necessary P_{i+2} with Q_{j+1}. A match in either permits us to again increment our weight and proceed from these new matched end points. Failing to match either pairing, forces us to conclude that at least two mismatched instructions must occur and so we again decrement our weight and compare P_{i+1} with $Q_{j+3;,}$ P_{i+2} with $Q_{j+2;,}$ and if necessary P_{i+3} with Q_{j+1}. Generalizing having concluded that at least *n* instructions must be skipped in the matching process, we compare in some implementation order (subject to the instructions existing) every possible matching of P_{i+1+k} with $Q_{j+1+n-k}$ for $0 \leq k \leq n$. For example, this algorithm will match the solid edges shown in Figure 1.

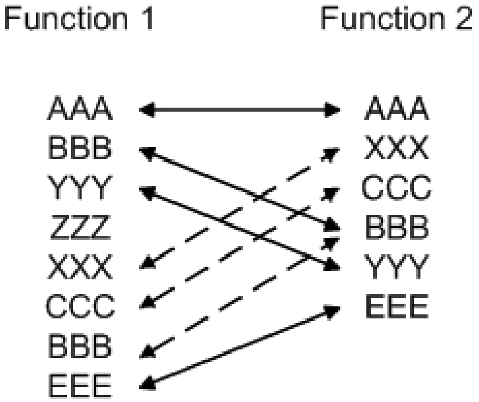

Figure 1. Possible matchings

A hill climbing algorithm may be invoked following execution of the greedy algorithm. Every possible way in which instructions from the one discovered clone might be matched with the other is computed, and these edges between the two clones not currently matched become candidates to be matched. To some user-specified maximum, in increasing order of the number of currently matched edges which block an edge from being matched (either as consequence of crossing it or of sharing an end point with it) all such blocking edges are marked unmatched, and the candidate edge together with all other now unblocked edges are marked matched, whenever this change increases the number of matched edges. Thus this algorithm will replace two of the solid edges shown in Figure 1, with the three dotted edges shown.

The hill climbing algorithm terminates when any of four conditions hold: when insufficient memory exists to implement it; when a user-specified number of iterations have been performed; when a given time limit is exceeded; or when no further improvement seems possible.

Iteration between the greedy and hill climbing algorithm continues as long as improvement in the matching between discovered clones occurs. Once no further improvement seems possible, matched forward branches which disagree as to the nearest recognized matched instructions at or prior to the addresses branched to are unmatched.

The subsequences discovered in the above process are upon conclusion each truncated so that either the last matched instructions within each determined subsequence, or alternatively the last match for which the weighting was maximal, becomes the last instruction within each respective clone.

The primary benefit of the above algorithm is that it is fast, simple, and appears to be effective. This algorithm when presented with longer sequences of instructions that match becomes correspondingly more forgiving of later instructions that do not, which seems reasonable (even desirable) behavior. Modification of the parameterized weights permits exact (type 1) matching of sub-sequences to be enforced, or more or less tolerance of mismatches.

Our algorithm is similar to that proposed by Baker [1], but it differs in that it aggressively seeks long clones that potentially contain mismatched code, rather than in aggressively seeking fragments of code that under potential parametric transformation match exactly, which can then potentially become part of longer clones that include mismatched code. It also differs in how it avoids reporting clones derived from overlapping instruction sequences.

Runtime complexity for the greedy algorithm is O($n*m*p^2$) where n is the total number of assembler instructions examined as candidates to begin a clone, m is the average matches in a discovered clone, and $p \geq 1$ is the average number of steps needed to find a match, in the presence of mismatched instructions.

Runtime complexity for the hill climbing algorithm is dominated by the need to consider *O(n*m)* possible edges between clones of length n and m, and worse the *$O((nm)^2)$* intersections that can arise between these edges. As consequence this algorithm cannot reasonably be expected to run to normal conclusion, or be able to obtain sufficient memory to run at all, when presented with very long clone pairs. However it can be expected to make improvements while running.

We are currently evaluating our tools empirically against several source-based clone detection tools.

5. Conclusions

We have presented a novel search based approach for performing clone detection that complements other available approaches. This approach permits one to perform clone detection on precisely those source files actively used in a build process, and to analyze source on which all the normal preprocessing involving macro expansion and header file inclusion has been performed. It offers the potential of building hybrid clone detection algorithms that can exploit commonality in the source code and the assembler produced from it, since source code, assembler, and the correlation between the two are all known. By exploiting the ability to interpret assembler instructions one can aggressively perform clone detection, not merely by using the available syntactic information, but also by considering the actual runtime behavior of the assembler.

Our clone detection software is reasonably fast, and produces comprehensive results that can be viewed through a web browser, a graphical visualization tool, and/or other back end clone evaluation software. While it must be customized for each assembly language that it is to be used on, this is a one-time cost.

Source code for our Java (JCD) and Assembler (ACD) clone detection tools are available [11].

Acknowledgments

This research is supported by grants from CA Canada Inc. and NSERC.

References

[1] Baker B. S. On finding duplication and near duplication in large software systems. *Proceedings of the 2nd Working Conference on Reverse Engineering*. **1995**

[2] Baker B. S. & Manber U. Deducing similarities in Java source from bytecodes. *USENIX Annual Technical Conference.* **1998**

[3] Davis I. J. & Godfrey M. W. Clone Detection by Exploiting Assembler (position paper). *Fourth International Workshop on Software Clones***.** **May 8, 2010,**

[4] De Sutter B., De Bus B., De Bosschere K. Time Binary Rewriting Techniques for Program Compaction. *ACM Transactions on Programming Languages and Systems,* **27(5) September 2005**

[5] Harman M, The Current State and Future of Search Based Software Engineering. *Proceedings of the 29th International Conference on Software Engineering*, **20-26 May 2007**

[6] Kapser C. J. & Godfrey M. W. Supporting the Analysis of Clones in Software Systems. *Journal of Software Maintenance and Evolution*. **Vol. 18(2) March 2006**

[7] Kapser C. J. Towards an Understanding of Software Code Cloning as a Development Practice. *PhD Thesis* **June 2009**

[8] Norman M. Clone Detection applied to Java Bytecode *Carleton University* (unpublished) **10 Dec 2008**

[9] Roy C. K. & Cordy J. R. A Survey of Software Clone Detection Research. Technical Report 2007-541 *Queens University.* **September 26 2007**.

[10] Selim G. M. K. Foo K. C. and Zou Y Enhancing Source-Based Clone Detection Using Intermediate Representation *17th Working Conf. on Reverse Engineering* **October 2010**

[11] SWAG: Software Architecture Group. www.swag.uwaterloo.ca