Similarity in Programs

Andrew Walenstein[1], Mohammad El-Ramly[2], James R. Cordy[3], William Evans[4], Kiarash Mahdavi[5], Markus Pizka[6], Ganesan Ramalingam[7], Jürgen Wolff von Gudenberg[8], and Toshihiro Kamiya[9]

[1] University of Louisiana at Lafayette, Center for Advanced Computer Studies,
P.O. Box 44330, Lafayette, LA 70504-4330, U.S.A.
walenste@ieee.org

[2] University of Leicester, Department of Computer Science,
University Road, Leicester, LE1 7RH, England
mer14@le.ac.uk

[3] Queen's University, School of Computing,
Kingston, ON, Canada, K7L 3N6
cordy@queensu.ca

[4] University of British Columbia, Dept. of Computer Science,
2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada
will@cs.ubc.ca

[5] King's College London Department of Computer Science,
Strand, London, WC2R 2LS
kiarash.mahdavi@kcl.ac.uk

[6] Technische Universität München, Institut für Informatik
Boltzmannstr. 3 85748 Garching, Germany
pizka@in.tum.de

[7] Microsoft Research India,
196/36 2nd Main, Sadashivnagar, Bangalore 560 080, India
grama@microsoft.com

[8] Julius-Maximilians-Universität Würzburg Lehrstuhl für Informatik II,
Am Hubland, D-97074 Würzburg, Germany
wolff@informatik.uni-wuerzburg.de

[9] National Institute of Advanced Industrial Science and Technology (AIST),
Information Technology Research Institute, Ubiquitous Software Group

Abstract. An overview of the concept of program similarity is presented. It divides similarity into two types—syntactic and semantic—and provides a review of eight categories of methods that may be used to measure program similarity. A summary of some applications of these methods is included. The paper is intended to be a starting point for a more comprehensive analysis of the subject of similarity in programs, which is critical to understand if progress is to be made in fields such as clone detection.

Keywords. computer programs, similarity, code clone, software comparison, program metrics, Levenshtein distance, parameterized difference, feature space, shared information, plagiarism, compression

Dagstuhl Seminar Proceedings 06301
Duplication, Redundancy, and Similarity in Software
http://drops.dagstuhl.de/opus/volltexte/2007/968

1 Introduction

Programs are compared for similarity in many contexts, including: detecting duplicate or cloned code within programs [1], detecting plagiarism, copyright or patent infringement [2], and removing redundancies for compression [3]. In such problem contexts the concept of similarity is fundamental concern. But what, precisely, *is* similarity between programs and how does one go about defining or measuring it?

This paper provides an overview to answers to such questions. The genesis of this overview was the discussion in a breakout discussion session at the Dagstuhl seminar on Duplication, Redundancy, and Similarity in Software [4]. The overview is divided into three separate threads of inquiry regarding similarity: what it is, how to measure it, and how to select a measurement instrument? Although the overview is brief, it may prove useful as an introduction to the topic, or as a seed for a more complete investigation.

2 Types of Similarity

One of the main points of discussion in the breakout group was a debate about what could be meant by "similarity," and the types of similarity that might be considered in software. Significant debate on the topic is even possible because the concept of "similar" appears to be, by its nature, imprecise. Even though no explicit consensus on the core definitions was raised, the group did appear to share a tacit common understanding of the term. Dictionaries tend to define "similarity" in terms of resemblance, interchangeability, or some manner of qualifying minor deviations in features or properties. Such definitions apply also to software in that the essential aspect of program similarity is that the degree to which two distinct programs are similar is related to how precisely they are alike. Vague, to be sure, but it was a workable common understanding.

The discussion on different types of similarity was far-ranging. It was not clear that the group had a clear notion of what constituted a different similarity "type". There appears to be different "kinds" of similarity that may be defined according to the method used to compare items. For instance, it is common in geometry to adopt a specific and precise definition that two triangles are *similar* when their angles are the same. A child, however, might note that two triangles are similar because they are both blue. These are different "kinds" of similarity, but are they also necessarily different "types"? Or are there kinds of similarities that fall into distinct categories and thus grouped into classes or types?

Unfortunately, this question was not directly debated in the breakout session. In order to clarify the overview we shall take cues from established notions of what a "program" is. Having a clean definition is critical since it is logical to assume that only when the definition of "program" is nailed down can one hope to properly pin the notion of similarity in programs.[1] The tacit consensus in the group tended to mirror the dominant epistemic philosophy in computer science's academic circles, namely, that of a syntactic/semantic dualism where semantics is understood in terms of some type of mathematical realism [5]. The discussion generally followed this philosophy, and it seemed the different types of similarity being discussed fell in line with the dual between *representational* and *semantic* or *behavioral* similarity.

2.1 Representational Similarity

Usually the term "program" refers specifically to the representational form which is commonly a sequence of characters forming a more complex text structure. Similarity can be defined in terms of the form, properties or characteristics of this representation. We distinguished between *textual*, *syntactic*, and *structural* similarity. While these are all familiar terms, the group did not spend time on disentangling the various meanings of the terms, and how they might apply. It was noted that similarity in the representation admits of similarity along different "levels" of abstraction, which correspond to the levels of abstraction at which one is able to view programs. For example, one could examine similarity at statement, block, class, unit, or architectural levels.

2.2 Semantic or Behavioral Similarity

Even though the term "program" is frequently identified with the representational form, the semantic interpretation of programs is not easily forgotten. The group had no trouble accepting that in at least some comparison models, two programs can be considered similar even if the main similarity is in their semantics (the functions they implement, for example).

The consensus of the group was that there are few intuitive ways to compare programs semantically unless the semantics are first translated (i.e., represented) into some other representation. For instance, once a program's semantics are written in `Z` notation, the (text of the) `Z` representation can be compared for similarity. It should be noted that semantic non-comparability is an important part of a debate on code clones. One side of the debate has argued that generally one does not wish to consider code chunks as clones if they are not complete blocks. The reasoning is that only complete blocks will have well-defined semantics. Nonetheless, the concept of similarity is more encompassing than code clones, and several types of semantic interpretations were considered. Discussion of most of these was accompanied by a debate on the the types of representations that might be used in its comparison. These types included:

- **Functional Similarity.**
  Intuitively speaking, (terminating) programs implement functions, so that

[1] For simplicity we will talk about "programs" and "software" as whole units but in most cases it is possible to generalize the discussion to talk about fragments or portions of programs.

two programs can be called similar if they implement a similar function. Of course, this definition merely replaces a mystery with an enigma: what constitutes a "similar function"? One suggestion made at the breakout session was that functional similarity can be described in terms of similarity in the input-output relationship. This type of similarity appeared to be one of the types that might be defined mathematically, as the concept appears to be related to mathematical approximation, which is a subject with a deep literature.

- **Execution Similarity.**
  This type of similarity is concerned with the sequence of execution of program statements, for example, Java byte code or assembly code. Similarity in execution will be related to similarity in the form of the program, since similarity in form would be required to find a correspondence in executable statements.

Missing was a more lengthy exploration of the types of different semantics or notions of behavior. For example, there are a variety of known types and systems of program semantics (denotational, operational, logics, automata, Petri-Nets, function theory, type theory, etc.). Each of these may have specific ways of stating how two programs may be alike yet not precisely the same. Metrics-based similarity was also posed as a different *type* of similarity, however it appears to be the case that metrics are either defined on the form or semantics and therefore can be classified as a particular way of *measuring* those types of similarity, not a type of similarity itself (see below).

3 Measuring Similarity

Once one admits that there may be different degrees of similarity, it becomes natural to ask for models for measuring the degree of similarity (either qualitatively or quantitatively), as well as how to collect the measurements (e.g., objectively or subjectively). This section provides an overview of different ways, both for syntactic and semantic similarity.

3.1 Syntactic (Representational) Similarity

- **Textual Similarity.**
  Measures such as Levenshtein distances, longest common sequence, and parameterized difference fall into this category (for a review, see Koschke [1]).
- **Metrics Similarity.**
  Metrics similarity is based on the comparison of the values of code metrics for two code fragments [7]. Code metrics inherently map features of the code or its structure into the domain of real numbers, which then have time-honored ways of defining similarity on them.

- **Feature-based Similarity.**
  By "feature-based" similarity we mean to include all ways of measuring similarity by the amount of correspondence there is between unordered lists of aspects or properties that can be identified in programs. In this sense the term "feature" is used much as it is in machine learning and text processing. For example, one may consider the list of identifiers used in two programs to be features. Then similarity can be measured by noting the amount of correspondence between the features. For another example, features called "n-grams" may be extracted from the text of the program, and the comparison might weight the features according to some notion of significance [8]. An interesting question brought up in the discussion is whether a metric can be considered a feature. The group concluded that it could.

  In the discussion there was a suggestion made that *feature-based* similarity might be considered a distinct *type* of similarity is because it could be aligned with a different epistemic philosophy of programs, namely an Aristotelian one [6]. One of the principal components of Aristotelian philosophy posits that it is the intellectually "graspable" aspects of things that define what type of thing they are, and thus provide the basis for comparison of differences and similarity. One could then use definable "features" as the basis for defining similarity in programs, where "features" correspond to their "graspable" essential aspects. At the present time it is not clear that such considerations lead to a truly distinct type of similarity.

- **Shared Information.**
  Programs can be compared using the methods of Shannon's information theory. The essential idea is to consider the two programs as text messages. If the programs are independent, then the amount of information conveyed in their concatenation will be proportional to the total length of their concatenation. In the case where one program is identical, then the extra copy only adds one bit of information to the message when it is concatenated. Logically, then, the amount of similarity between two programs will therefore be related to the amount of information they share. Schemes of approximating this similarity (which is incalculable) include the *normalized compression distance* measure [9].

3.2 Semantic or Behavior Similarity

As was mentioned previously, the group felt that it was generally difficult to find measures of semantic similarity. What is often done instead is to try to find some representation or approximation of the semantics and define similarity on these. Some potential ideas for measuring semantic similarity are:

- **Execution Curve Similarity:** execution traces can be plotted as a curve by mapping program location to one spatial dimension and time on a separate (e.g., Lu *et. al* [10]). Then execution traces can be compared for similarity by measuring the similarity of the resulting curves. Preprocessing can be done using curve normalization in order to compare different execution curves using curve fitting. When the similarity in question is self-similarity, execution graphing techniques might still be applied [11], and the issues become related to issues such as optimization and branch prediction.

- **Input-Output Relation Similarity:** measuring the fraction of inputs for which two programs produce the same output. More complicated notions of input-output correspondence need to be contemplated when the input or output spaces are not finite.
- **Semantic Distance:** measuring the cost of mutating one program to another. The essence of the idea is to apply the concept of so-called Levenshtein differences in the semantic space. As with any such distance, one needs a set of mutation operations and their costs. Such might be conceivable with a set of program mutation operations, some of them semantics-preserving (e.g., loop unrolling) and some of them semantics non-preserving (e.g., deleting a program statement).
- **Abstraction Equivalence Distance:** the level of abstraction that one needs to reach before two programs are identical. Abstraction is the process of removing (inessential) details. The level at which two different programs become the same is a measure of how different they were in the first place.

  It is worth mentioning that this type of test is actually used in real life in court cases comparing programs for copyright infringement. The basis of case law in certain circuits of the American court system, for example, is the so-called "abstraction-filtration-comparison" test [12]. In this context, there are two important aspects to the logic behind the test. The first is to find the level of abstraction at which the idea can be separated from expression. The second is to acknowledge the fact that two different expressions can nevertheless (falsely) appear identical if abstracted to too high of a level.
- **Program-dependence Graph Similarity.**
  Comparing the program dependence graphs of two programs using some sort of graph comparison measure. In this case, the structure of the program is a proxy for the semantics.

4 Measuring: For What Purpose?

Application will normally dictate what type of similarity to use, and the particular measure to use. For some applications, syntactic similarity is the most obvious base for comparing code. In other applications it is semantic similarity. Yet in other ones, it is a combination of both. From our discussions, the following lists some possible applications for different types of similarity.

Semantically equivalent clones are those code segments that are have different representations (text) but they mean the same thing, meaning one can be replaced by the other. Finding these types of clones is useful in applications of program compression [3] and re-engineering. Current clone detectors will not be able to find these in the general case, however they may be helpful in finding that sub-class that are similar syntactically as well as semantically equivalent. This sub-class is also useful to find for the purpose of refactoring.

Syntactically identical but semantically distinct clones might be useful to find in the case where refactoring can be applied to unify the differences. Perhaps the canonical example is the use of parametric polymorphism to refactor two similar functions on different types. This class of clones might also be useful in program compression if the different semantics are unimportant in the specific program context and can be factored out.

Execution curve similarity may be useful to evaluate to see if a program is an obfuscation of an original, or to compare versions of a program to spot an introduced bug.

5 Conclusions

Program similarity is a core concept in areas such as clone detection, program evolution, intellectual property analysis, program maintenance, program compression, and plagiarism detection [2]. There are many different notions and measures for software similarity yet, while we are aware of reviews of clone detection techniques, we have not yet encountered any work that has sought to clearly and comprehensively define the essential notions of similarity and to map out an organized space of the possible types and methods. This initial overview, while short, has indicated that there are many deep questions in the area and that there may be several ties to important topics in many areas of computing science. We expect that improved analysis of the types and methods of software similarity will lead to new and improved methods for comparing code.

References

1. Koschke, R.: Survey of research on software clones. [4] ISSN 1682–4405.
2. Walenstein, A., Koschke, R., Merlo, E.: Duplication, redundancy, and similarity in software: Summary of Dagstuhl seminar 06301. [4] ISSN 1682–4405.
3. Evans, W.: Program compression. [4] ISSN 1682–4405.
4. Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, Dagstuhl, Germany, Dagstuhl (2006) ISSN 1682–4405.
5. Maddy, P.: Realism in Mathematics. Oxford University Press (1990)
6. Rayside, D., Campbell, G.T.: An aristotelian understanding of object-oriented programming. In: OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2000) 337–353
7. Kontogiannis, K.A., Demori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. Applied Categorical Structures **3** (1996) 77–108

8. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware: "Vilo" method for comparing and searching binary programs. In: Proceedings of BlackHat DC 2007. (2007) https://blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf.
9. Chen, X., Francia, B., Li, M., Mckinnon, B., Seker, A.: Shared information and program plagiarism detection. IEEE Transactions on Information Theory **50** (2004) 1545–1551
10. da Lu, C., Reed, D.A.: Compact application signatures for parallel and distributed scientific codes. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–10
11. Koike, H.: The role of another spatial dimension in software visualization. ACM Trans. Inf. Syst. **11** (1993) 266–286
12. Lemley, M.A.: Convergence in the law of software copyright? Bekeley Technology Law Journal **10** (1995)