Deriving Categories of Semantic Clones from a Coding Contest

Torsten Görg
torsten.goerg@itemis.de
itemis AG

Abstract: *This paper establishes subcategories of semantic code clones, based on code from a coding contest. We provide these clone subcategories as a basis of a benchmark for semantic code clone detection algorithms.*

1 Introduction

Beyond code clones that result from copy and paste activities, another reason for clones is independent implementation of the same abstract algorithm. These so called mental clones or semantic clones realize the same or at least a similar functionality but might look quite different in their syntactic structure [1]. It is not possible to detect all semantic clones automatically because the semantic equivalence of code fragements is not decidable in general. Nevertheless, it is worth to build clone detectors that approximatively find as many of these clones as possible.
We have created several prototypical detectors for semantic clones. Even some advanced kinds of variations are handled by our detectors, e.g., the reordering of statements in a statement sequence [2] and the extraction of loop invariant code. Nevertheless, the detection results do not meet our expectations. Either our detection approaches are not appropriate for real semantic clones or the analyzed system do not encompass as many relevant semantic clones as we expect. To improve our clone detectors we need a deeper understanding of this problem.

2 Alternative Development Process

A usual way to create clone detectors is to construct a detection mechanism in theory, implement it, and evaluate the clone detection results for several example systems. This approach has several disadvantages. During the creation, it is not guaranteed that the detection mechanism really covers relevant semantic clones. On the other hand, the implementation of a clone detector might cost much effort. This is especially true for semantic clone detectors because more sophisticated mechanisms are required than for the detection of copy and paste clones. Overall, the approach is contrary to some major software engineering principles. I.e., the analysis and specification of requirements is missing. Detailed requirements for a clone detector should describe how the clones look like that are most important to detect.
A major problem is that in most cases the code clones in a given system are not know in advance. An oracle is missing. As a consequence, for the construction of the detection mechanism you can only guess how the clones might look like in real systems. Especially for semantic clones, many different variations between the matching code fragments are possible. It is not obvious which kinds of variations are more frequent than others. There is a high risk to spend much development effort for the handling of irrelevant kinds of variations.
Our idea is to turn the process around in order to attack the clone detection problem from another side. In a first step, we explore a training set of example systems manually in order to learn about the semantic clones in these systems and the kinds of variations that occur there. Then we implement clone detectors for the most frequent kinds of clones to minimize development costs. In this paper we focus on the manual exploration step.

3 Code Variations Modeling

To specify which kinds of semantic clones are most relevant, we model them as subcategories of the semantic clones category (clone type 4). Clone categories should be as independent from any detection technique as possible. Although it is an open question if static or dynamic approaches are more appropriate for the detection of semantic clones, subcategories of semantic clones cannot be defined based on the input output behaviour and a black box view of the analyzed code fragments, as the input output behaviour of semantic clones is identical, by defintion. We have to take the inner structure of the code into account.
We view the differences between semantically matched code fragments as structural variations that do not affect the semantics significantly. More precisely, the differences between two code fragments are expressed as a sequence of locally applied, fine-grained code variations. We define subcategories of clone type 4 as abstractions of such variation sequences. A subcategory is given by a sequence of variation steps, where each step is a fine-grained variation of a particular variation kind.

4 Manual Code Exploration

For arbitrary example systems, code clone oracles are not available. But we can make use of coding contests, which are conducted quite often nowadays, e.g., by Google. A coding contest provides a programming task that is solved by many submitters. For some contests, the set of all correct solutions is published. The correctness of the solutions guarantees that they are all semantic clones of each other.

Wagner et al. [3] used the results of Google Code Jam in 2014 to investigate coarse grained categories of semantic code clones and to create a benchmark for clone detectors. They defined five categories: variations in algorithms, in data structures, in input/output operations, in the use of libraries, and in the object-oriented design. Each code example in their benchmark is assigned to exaclty one of these categories. We have inspected the examples in detail and have seen that each clone pair encompasses multiple fine-grained variations of different kinds. Our identification of kinds of fine-grained variations is based on the code examples in the benchmark of Wagner et al.

We have explored each clone pair of the benchmark manually. Fine-grained variations are applied to the code fragments until both sides of a clone pair are identical. The order of the variation steps is not necessarily unique. But different sequences often contain variations of the same kinds. Finally, we derive clone subcategories as abstractions of the calculated variation sequences.

5 Variation Kinds Catalog

To prepare the manual code exploration we gather an initial set of possible kinds of fine-grained variations, in advance. This catalog provides us code patterns to look for during the exploration. If none of these patterns can be applied we have to add further kinds of variations to the catalog, as an adaptation of our theory to the real world program code. Without our new approach, clones based on these kinds of variations would be missed.

The initial catalog encompasses syntactical variations, data flow related variations (including data flows via pointer dereferencing and side effects), control flow related variations, differently cut procedures, use of different data structures and algorithms, and local implementations instead of library function calls.

6 Evaluation

The following table lists the most important kinds of variations we have found in the analyzed benchmark code, together with their absolute and relative frequencies:

variation kind	abs	rel
consistent identifier renaming	36	18%
elimination of unnecessary code	32	16%
adapt string literal	21	11%
replace printf/scanf by fprintf/fscanf	15	8%
type widening	14	7%
expression transformation	9	5%
replace while-loop by for-loop	7	4%
eliminate intermediate result variable	7	4%
restructure if-then-else branch cascading	6	3%
loop index range shift	5	3%
exchange of mathematical algorithm	5	3%
extract assignment	5	3%
reorder statements	4	2%
merge of if-then-else-cascade branches	4	2%

7 Conclusion

Now we have a basis at hand that guides our further development of semantic code clone detectors. To detect a clone of a particular subcategory, handlings of all variation kinds in its sequence have to be implemented.

The results are derived from artifically provoked semantic code clones. We cannot conclude that real systems encompass such clones as well.

Furthermore, the solutions of the coding contest might be influenced by the task descriptions and the given hints.

Nevertheless, the situation of independent coding is realistic. If there are semantic clones in real systems, they probably look similar to the clones found in this study.

References

[1] Chanchal Kumar Roy and James R. Cordy, “A survey on software clone detection research,” technical report, Queen’s University, Canada, 2007.

[2] Torsten Görg, “Interprocedural PDG-based Code Clone Detection,” in Proc. of the 18th Workshop Software Reengineering & Evolution (WSRE 2016), Bad Honnef, Germany, 2016.

[3] Stefan Wagner et al., “How are funtionally similar code clones syntactically different? An emperical study and a benchmark,“ PeerJ Computer Science 2:e49, 2016.