Interprocedural PDG-based Code Clone Detection

Torsten Görg
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
torsten.goerg@informatik.uni-stuttgart.de

Abstract: *This paper suggests a PDG-based code clone detection algorithm that handles procedure calls with summary information about the called procedures in order to improve the precision of the detection results.*

1 Introduction

One possible approach for code clone detection is using PDGs (Program Dependency Graphs) as intermediate representation [1]. In contrast to other program representations like token streams or abstract syntax trees, PDGs provide a clone detection that is more closely related to the semantics of the analyzed program and provides more precise results. For each clone pair candidate, our approach compares two graph backward slices in the PDGs to decide about the clone property. This reduces clone detection to a graph reachability problem with additional equivalence constraints. Our goal is to detect any PDG subgraph matches even if a subgraph is in the middle of a procedure's PDG.

Because of the runtime complexibility of PDG-based clone detection that is generally cubic in the total number of nodes in all PDGs, in the worst case, one has to handle performance aspects carefully. Especially interprocedurality might threaten an acceptable performance. When the slice matching process reaches procedure call nodes and the called procedure is the same for both slices, the procedure call nodes are obviously equivalent. The case when different procedures are called is more complicated. Conservatively, a procedure call matches any other call, producing many false positives. The opposite rejects any calls of different procedures, resulting in false negatives.

A more precise implementation could dive into the details of the called procedures and try to match their bodies for each call. As a procedure is usually called from multiple call sites it would be analyzed repetitively. In the worst case, exponentially many repetitions would be necessary. To avoid this complexity, we calculate summaries instead, following the idea introduced by Reps, Horwitz, and Binkley [2] for interprocedural slicing. They suggest to calculate summary edges that indicate the set of procedure input parameters a procedure output value is possibly dependent on. Our first approach was to implement exactly this in our clone detector. It models reads and writes on global variables and reference parameters as well as return values as artificial parameters in the intermediate representation. For matching two procedure call nodes that reference different procedures, it looks up the summary edges of these procedures and tries to match the resulting sets of dependencies.

The problem with this approach is that it does not provide any information how to map dependencies in the first set to dependencies in the second set. But this information is crucial to continue the matching process behind the procedure call nodes. As an approximation, the matching might assume that the formal parameters are declared in the same order for both procedures and sacrifice the detectability of parameter reordering. But for reads or global variables, this assumption is unrealistic.

Our solution is to calculate clones in two passes. Pass 1 is coarse-grained and finds clones that span procedures from their output to their input. Pass 2 is fine-grained and detects any PDG subgraph matches. Summaries of the procedure clones found in pass 1 are used to facilitate finding further procedure clones in pass 1 as well as fine-grained clones in pass 2.

2 Procedure Clone Summaries

More exactly, our procedure clones do not necessarily cover whole procedures. For each discriminable component of all output values of a procedure, a separate backward slice is calculated. We call these slices features. In general, a procedure implements multiple features. I.e., an output value of a record type provides separate features for all record components. The elements of an array are viewed as one component with just one slice. A feature is uniquely characterized by the output value component that spans it. The backward slice of a feature is limited by references to input value components of the procedure that contains the feature. The component granularity of input values is the same as described above for output values.

Pass 1 of our clone detection compares each feature with the features of all other procedures. We heuristically assume that a feature is not a clone of

another feature of the same procedure. If the slices of two features completely match, a feature clone pair is recognized. During the matching process, irrelevant nodes are skipped wherever possible. The clone detector stores a summary of each detected feature clone pair. Such a summary encompasses references to the output value components that characterize the features and a description of the mapping between corresponding input value components.

In comparison to procedure clones that have to cover whole procedures, the feature clones approach is more flexible and provides a higher probability to find clones at that coarse granularity level. Nevertheless, for many software systems, we expect only a small number of feature clones. But for the subsequent fine-grained clone detection in pass 2 not only the recognized feature clones are relevant. The negative information that two features are not in a clone relationship is also useful. It can be used to eliminate clone candidates that call these features.

3 Unification of Clone Parameters

A full structural congruence of the slices that represent code fragments is not sufficient to decide about the clone property. Additionally, it has to be checked if corresponding input values are consistently referenced. E.g., x + x * y is congruent with a + b * b but not semantically equivalent. This is exactly the difference between type 2 clones and parameterized clones, as defined by Baker [3]. We have integrated a parameter unification in pass 1 that enforces a bijective mapping between the input value components of the matched code fragments.

A weaker but still sufficient check allows us to map an input value component of one fragment to a literal value in the other fragment. E.g., x + 3 * y matches a + b * c, although these fragments are not semantically equivalent. The second fragment is more general than the first.

A similar situation is a mapping of some input value components of one fragment with multiple input value components of the other one. E.g., x + y * z matches a + a * a. Here, the first fragment is more general than the second. This kind of clone is called surjective clone, because a surjective mapping between the input value components of both fragments exists.

4 Evaluation

As an example, we analyse the program gnuplot_x11 from the gnuplot open-source software package. It consists of 75 procedures. A procedure's average number of features is 20.1, resulting in #feat = 1570 features overall. The number of feature clone pair candidates is #cand = 1139859, #cand < $\#feat^2$ = 2464900. #cand is less than half of $\#feat^2$ because of our exclusion of feature clones within the same procedure.

Only 42 feature clone pairs are found. This low number of feature clones confirms our expectation. Furthermore, all detected feature clones are rather small. Except of one clone of size 48, the clones do not encompass more than 7 nodes.

To evaluate pass 2 we have compared the clone sizes histograms for fine-grained clones without and with usage of feature clones summaries. The usage of summaries shifts the histogram to smaller clones because more matches of call nodes are excluded. This splits clones into smaller ones.

5 Future Work

It could be an appropriate heuristic to assume that features of a procedure p_1 are usually not clones of features of a procedure p_2 if p_1 calls directly or indirectly p_2. This constraint can be checked by traversing the call graph in topological order. We have to validate this assumption with further experiments.

Our current approach supports the matching of code fragments in situations where procedure call nodes occur in both code fragments in corresponding positions. Another interprocedural situation is matching a procedure call with equivalent inlined program code. To support this kind of clone as well, a more comprehensive preparation phase is required. It is not sufficient to compare feature slices with other feature slices. They have to be compared with slices spanned at arbitrary positions inside of procedures. This can be calculated by another pass before pass 2. It has to be evaluated if it is worth to spent the additional effort in contrast to an implementation that dives into the details of the called procedure in such situations.

References

[1] Raghavan Komondoor and Susan Horwitz, "Using Slicing to Identify Duplication in Source Code," in Proc. of the 8th International Symposium on Static Analysis (SAS '01), London, UK, Springer-Verlag, 2001.

[2] Thomas Reps, Susan Horwitz, and David Binkley, "Interprocedural slicing of computer programs using dependence graphs," in Proc. Of the ACM SIGPLAN 1988 conference on Programming language design and implementation (PLDI '88), New York, NY, USA, ACM, 1988.

[3] Brenda S. Baker, "On finding duplication and near-duplication in large software systems," in Proc. of the Second Working Conference on Reverse Engineering (WCRE '95),Washington, DC, USA, IEEE, 1995.