Are Decomposition Slices Clones?

Keith Gallagher
Computer Science Department
Loyola College in Maryland
4501 N. Charles St. Baltimore, MD. 21210 USA
kbg@cs.loyola.edu

Lucas Layman
Computer Science Department
North Carolina State University
Raleigh, NC 27695-8206
lmlayma2@unity.ncsu.edu

Abstract

When computing program slices on all variables in a system, we observed that many of these slices are the same. *This leads to the question:* Are we looking at software clones? *We discuss the genesis of this phenomena and present some of the data observations that led to the question. The answer to our query is not immediately clear. We end by presenting arguments both pro and con. Supporting the affirmative, we observed that some slice-clones are evidently the result of the usual genesis of software clones: failure to note appropriate abstractions. Also, slice-clones assist in program comprehension by coalescing into one program fragment the computations on many different variables. Opposing the proposition, we note that slice-clones do not arise due to programmer intent or the copying of existing idioms.*

KEYWORDS: SOFTWARE MAINTENANCE, SOFTWARE COMPREHENSION, CLONE DETECTION, PROGRAM SLICING, DECOMPOSITION SLICING

1. Introduction

The detection of software clones is an important software comprehension and evolution activity. *What is a clone?* Baxter, et al., [2] say that a clone is "a program fragment that [is] identical to another fragment." The Detection of Software Clones website [1] declares that a clone "is a parametrized copy [in which] [v]ariable names and function calls have been renamed and/or types have been changed... or a copy with further modification [in which] [s]tatements have been added or removed." Krinke [9] uses the term "similar code." Ducasse, et al. [5] use the term "duplicated code," Komondoor and Horwitz [8] also use the term "duplicated code" and use "clone" as an instance of duplicated code. Mayrand, et al., [11] use metrics to find "an exact copy or a mutant of another function in the system."

Baxter, et al., [2] list some possible causes of clones:

- Code reuse by copying pre-existing idioms
- Coding styles
- Instantiation of definitional computations
- Failure to identify/use abstract data types
- Performance enhancement
- Accidents

Once clones are detected, they can be systematically removed by any number of methods. Parametrized function/method invocations and in-line replacement via macros are just 2 replacement techniques. The benefit to a software engineer involved in understanding, changing, reverse engineering, or refactoring code is evident. Multiple uses of cloned code do not need to be discovered then changed as one maintenance task; one abstraction gives insight to many apparently unrelated program computations.

We have constructed a graph depicting the relationship of decomposition slices of a program (in a process described below), and use this as a basis for our discussion of whether decomposition slices can help identify software clones. A program slice of program p on variable v, or set of variables, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed [13]. The pair (v, n) is called a *slicing criterion*. Slices can be approximated automatically on source programs by analyzing data flow and control flow. Surveys of program slicing may be found in [3, 4, 12]. A decomposition slice [7] does not depend on statement numbers. It is the union of a collection of slices, which is still a program slice [13].

2 Observations

We have some observed data, reported in another venue [6]. A brief reprise follows. A decomposition slice

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

captures all relevant computations involving a given variable. Computing the decomposition slice for each variable in the program forms a graph, using the partial ordering induced by proper subset inclusion. An edge from A to B means $B \subset A$ and there is no C, such that $B \subset C \subset A$. Figure 1 is the decomposition slice graph of a differencing program. It has 95 nodes and 364 edges.

Each node in the graph represents the decomposition slice on a particular variable, *v* . The node displays the name of the function the variable is in followed by the variable name. It also displays the "size" of the slice, which is the cardinality of the set of statements that constitute the slice. Larger sets will be toward the top of the graph; smaller sets will be lower; edges point downward. Every variable or programmer defined constant (`enum` or `typedef` value) generates a slice, as do unused global variables included in library header files. This is the cause of the "fan out" at the bottom of Figure 1. These decomposition slices do not have any executable statements. There are 29 of these "empty" slices. Removing them and the incident edges lowers the count to 66 nodes and 161 edges. (The resulting graph is not shown.)

To further reduce the visual clutter and make the graph more readable for the software comprehender, we output only one node for each equivalent decomposition slice, using simple set equality. That is, those variables with identical statements constituting their decomposition slices were represented by one node. Figure 2 shows the result of reducing the graph of Figure 1. The reduced graph of Figure 2 has 34 nodes and 43 edges. We also augmented the node information to show how many other nodes are equivalent (not readable in the Figure). The border of the node was also widened in proportion to the number of equivalent nodes. This Figure includes the "empty" slices at the bottom; their inclusion yields only one node in the graph.

Due to the vagaries of the layout algorithm, the reduced graph is rotated about the vertical axis with respect to the graph of Figure 1. The five upper leftmost nodes of Figure 1 are the five upper rightmost nodes of Figure 2. The three nodes to the upper right of Figure 1 are collapsed to the single node in the upper left of Figure 2. Following the edges from these nodes downward in Figure 1 leads to the "fan-out" in the lower center of the figure. This fan-out of 14 nodes is reduced to *two* nodes in Figure 2; and the node reduction induces a drastic reduction in the number of edges.

So in this small example, we reduce a graph of 95 nodes and 364 edges to one of 34 nodes and 43 edges, a reduction of 62% in the node count by merely noting that some slices *are the same.*

2.1 What Kind of Clones Are Found?

In the following simple case of cloning, a programmer copies and modifies a piece of code such as

```
for (i = 1; i < 10 ; i++ )
     a[i] = a[i] * a[i];
```

to

```
for (i = 0; i < 100; i++ )
     b[i] = b[i] * b[i];
```

The second fragment is easily obtained from the first by changing the numeric constant `10` to `100` and array variable name `a` to `b` The approach presented herein does not find these straight syntactic copies and changes. This is because our approach is slice-based, and program slices depend on variables. The clone detection techniques of Baxter [2], Krinke [9], Ducasse, et al. [5], Komondoor & Horwitz [8], Mayrand, et al., [11] would succeed on this fragment.

What we do find are equivalent slices of the following sort. The program slices at the last statement on variables `t` and `a` are the *same* in this in-place array transposition fragment. When two or more such fragments are located, it is displayed in the graph as one node.

```
for(i = 0 ; i < r ; i++) {
   for( j = i + 1 ; j < r ; j++) {
      t = a[i][j];
      a[i][j] = a[j][i];
      a[j][i] = t;
   }
}
```

Note, however, if the above swapping fragment is copied unmodified by cut-and-paste into another separate function, it would *not* be detected as a duplicate by this technique.

2.2 Application to "Clone Detection Experiment"

The Detection of Software Clones website [1] defines three clone types: (1) an exact copy: (2) a parametrized copy; (3) a copy with further modification. The `for-statement` examples above are type 3. Our process finds only a subset of the type 1 clones; we will call these *slice-clones*, to differentiate the technique by which the duplicates were found.

The *weltab* system is one of the **C** software suites available from the Detection of Software Clones website. It has 39 **C** source files that compile into 37 executables. Two files that do not have a `main` have 41 support functions that account for a total of 27 slice-clones. Table 1 shows the node reduction counts.

In order to do a statistical analysis of the data, we had to pare the sample population somewhat. Graph reduction

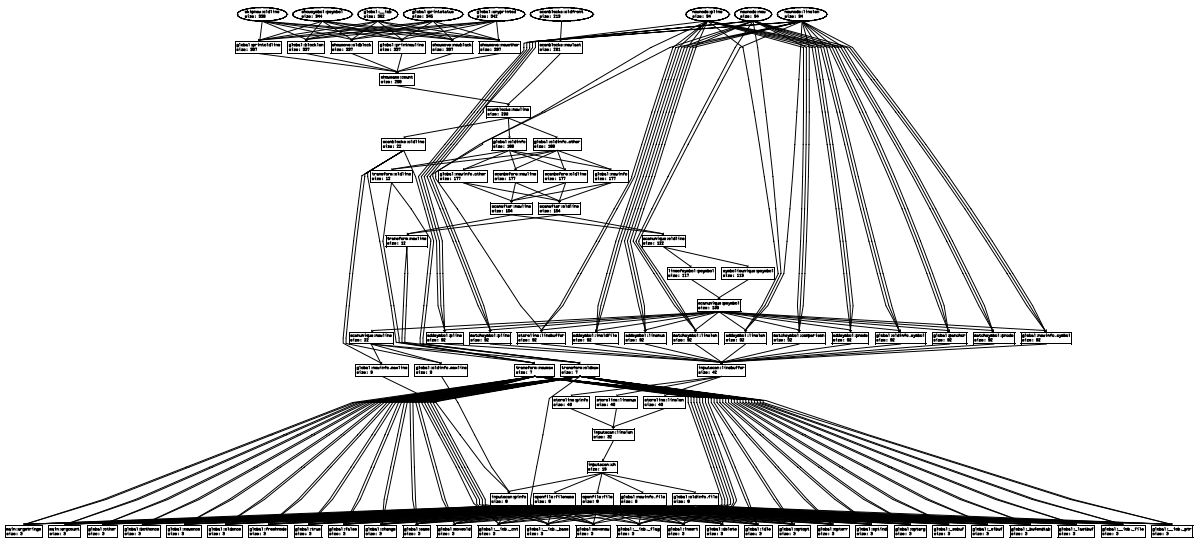

Figure 1. The decomposition slice graph of a differencing program.

data from a previous study [6] was combined with selected information from the *weltab* system shown in Table 1. In order to ensure a group of independent samples, we excluded the reduction information for files where the reduction numbers were duplicates of another file. For instance, all of the `r*tmp` were not included in the analysis since they contain identical node reduction numbers, but instead a single data set was chosen to represent the group. In cases where the node counts were very close, e.g. `canv` and `cnv1`, the source code was examined more closely (admittedly on a subjective basis) to see if the code was essentially the same with minor modifications. If they were essentially the same, the sample with the lowest percentage of node reduction was chosen as the representative. If there was some doubt as to the similarity between two code files, we chose to discard one (or more) of the node counts in order to prevent skewing of the results as much as possible. We received a final sample size of $N = 31$.

Performing a simple regression analysis on the number of reduced nodes versus the number of original nodes produces some interesting results. A strong correlation exists between the number of reduced nodes and the number of original nodes, giving us an R^2 value of 89.7%. We test the null hypothesis, $H_0 : \beta = 0$, to determine if the number of reduced nodes is independent of the number of original nodes using a *t*-test. We receive a p-value < 0.005, therefore we reject the null hypothesis and conclude that the two variables are, indeed, dependent. This conclusion, plus the high R^2 value, indicates that, by knowing the number of nodes in the original graph, we can predict with a fair amount of confidence an approximation of the number of nodes in the corresponding reduced graph.

We would also like to characterize the percentage of node reduction over the total population of decomposition slice graphs. To do so, we first perform a K-S Test to ensure the population is normal. Performing this test on the sample population, we receive a signficance value of .189. Since this value is greater than .005, we can conclude that the sample population does not deviate from a normal population. We now conduct our *t*-test. From this data, we were able to determine with 95% confidence that the true mean percentage of node reduction is between 50.0% and 60.3%.

A simple *diff* on the source files where the node counts were similar, such as the `r*tmp` files, shows that the actual differences between these "systems" is extremely small. The high number of equivalent source lines strongly suggests that these systems were created by cut, paste and change.

3 The Question

So the question becomes: *Are slices-clones clones?*

While admitting that these artifacts are identical, we go through Baxter's list of section 1, and add a few other ideas. We do not discuss coding style and performance enhancement, as they do not seem to apply in this context. We now present the pro and con arguments for slice-clones as software clones.

3.1 Pros

1. The slice-clones arose due to "instantiation of definitional computations."

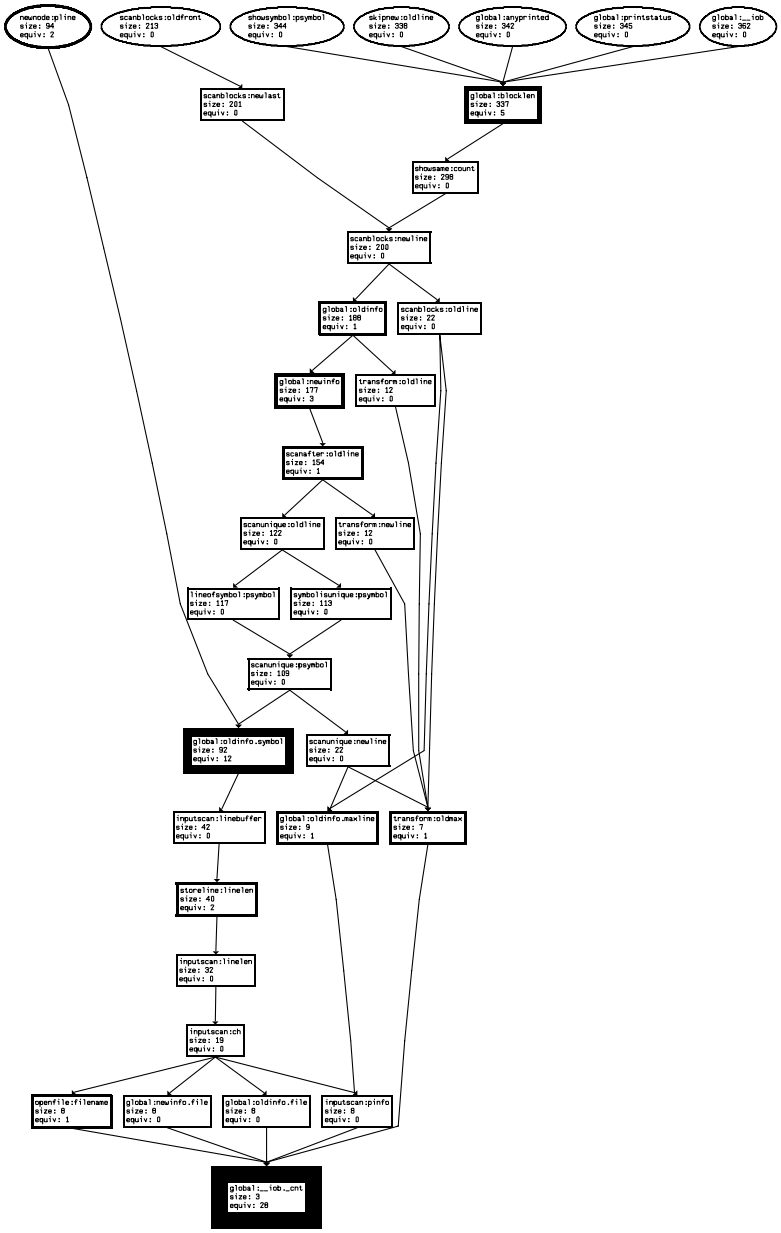

Figure 2. The reduced decomposition slice graph of Figure 1.

System	*Original*	*Reduced*	*Reduction*
cand	284	132	53.5%
canv	329	179	45.6%
ccibm	13	4	69.2%
ccibmx	13	4	69.2%
ccibmxp	13	4	69.2%
cnv1	329	176	46.5%
cnv1a	329	175	46.8%
cprint	16	9	43.8%
cumt	318	168	47.2%
ejcn88	353	184	47.9%
fixw	268	118	56.0%
lans	349	181	48.1%
lansxx	349	181	48.1%
linecoun	133	68	48.0%
ofca	281	139	50.5%
r11tmp	349	182	47.9%
r26tmp	349	182	47.9%
r51tmp	349	182	47.9%
rcvr	307	192	37.5%
rsum	349	182	47.9%
rsumxx	349	182	47.9%
sped	288	168	41.7%
spol	289	160	44.6%
totl	315	168	46.7%
unpr	293	128	56.3%
vedt	293	162	44.7%
vfix	293	162	44.7%
vful	289	140	51.6%
vset	195	85	56.4%
vtot	289	140	51.5%
xfix	293	162	44.7%

Table 1. weltab node reduction data

2. The slice-clones arose due to "failure to identify/use abstract data types."

3. The equivalent slices are not on the same variable.

4. Enhance the definition of a clone (or "duplicate code").

5. Noting similar sized analysis output leads to a straightforward detection of cloned systems.

Items 1, 2, and 3 coalesce. For item 1, we note that some definitional computations, such as the array transposition code above, involve many variables. Program slicing demonstrates the computational relationship between these variables in the embodied computation. This also supports item 2, in that these computational definitions are usually "abstractable" into functions or methods. Item 3 supplies more on-demand information for a comprehender. Suppose the comprehender considers a slice on a program variable. All other variables with equivalent computations are readily available. Such information is a by-product of our analysis. It would seem that this assuages the information overload. Combining these 3 items together yields a formidable abstraction mechanism for the comprehender.

For item 4, we argue that slice clones do indeed find duplicate code. While these duplicates are a proper subset of type 1 clones, they do offer insight into the *relationships* between clones. It would seem to be a good thing when a comprehender can identify relationships, as the relation creation in itself is an abstraction mechanism.

For item 5, we suggest that the data adds another complementary metric to those proposed by Mayrand, et al. [11] and is supported by two observations. This reduction technique finds artifacts that are "bad" in Mayrand's proposed relative evaluation scale; it does indeed find some exact copies. Also, the examination of the data table suggests other artifacts that are easily checked with *diff*.

3.2 Cons

The list of con arguments, while shorter, carries significant weight. Given that the current imperative and object paradigms are the only ones we have, we have to deal with them. They cannot be wished away.

1. Some obvious clones are not detected; the technique does not find all unmodified cut-and-paste clones in differing functions.

2. Slice-clones were introduced by "accident," not programmer intent.

3. Slice-clones did not arise due to "code reuse by copying pre-existing idioms."

4. The reduction in the number of nodes is statistically consistent and statistically significant.

Item 1 is extremely troubling, for it seems that the slice-clone technique does not perform the activity for which clone detection was introduced!

For items 2 and 3, the equivalent code fragments are accidental, not intentional. In fact, one could argue that these slice-clones are *essential* to effective software engineering. Given the tools we have, many variables and objects are "glued" together to form computational entities. It is precisely this activity that creates a system. This is not cloning; it is creating. These equivalent computations did not arise by *copying* pre-existing idioms; they arose by *creating* idioms.

Finally, the size and significance of the reductions argue that we are currently programming with the wrong idioms. For instance, a simple *swap* operation need not evidence an intermediate variable. Linger, et al. [10] suggested a simple `a, b = b, a;` for a variable swap in 1979. We need more simple and direct ways to embody "computational definitions." This is achievable with today's technologies. However, this is not an argument for object orientation. While objects give analysis and design insight, they are still *written* much like **C**. And slicing objects is not easy. This approach is also too fine-grained for today's gigantic, complicated and critical systems. Perhaps these new idioms, whatever they are, will not even be sliceable.

4 Conclusion

The intent of this paper is polemical. We have some data, and conversations with the researchers doing clone detection suggested the title.

The data is interesting and suggests both positive and negative responses to the resolution. We await the response of the community.

References

[1] Detection of software clones. http://www.bauhaus-stuttgart.de/clones.

[2] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance '98* ICSM-98, 1998.

[3] D. Binkley and K. Gallagher. A survey of program slicing. In M. Zelkowitz, editor, *Advances in Computers*. Academic Press, 1996.

[4] A. DeLucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[5] S. Ducasse, M. Reiger, and S. Demeyer. A language independent approach for detecting duplicate code. In *Proceedings of the International Conference on Software Maintenance '99* ICSM-99, 1999.

[6] K. Gallagher and L. O'Brien. Analyzing programs via decomposition slicing. In *Proceedings of International Workshop on Empirical Studies of Software Maintenance, WESS*, 2001.

[7] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[8] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[9] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001.

[10] R. Linger, H. Mills, and B. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Massachusetts, 1979.

[11] J. Mayrand, C. LeBlanc, and E.Merlo. An experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance '96* ICSM-96, 1996.

[12] F. Tip. A survey of programming slicing techniques. *Journal Of Programming Languages*, 13(3):121–189, 1995.

[13] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.