Identifying Similar Code with Program Dependence Graphs

Jens Krinke
Lehrstuhl Softwaresysteme
Universität Passau

Abstract

We present an approach to identify similar code in programs based on finding similar subgraphs in attributed directed graphs. This approach is used on program dependence graphs and therefore considers not only the syntactic structure of programs but also the data flow within (as an abstraction of the semantics). As a result, there is no tradeoff between precision and recall—our approach is very good in both. An evaluation of our prototype implementation shows that our approach is feasible and gives very good results despite the non polynomial complexity of the problem.

1 Introduction

Duplicated code is common in all kind of software systems. Although cut-copy-paste (-and-adapt) techniques are considered bad practice, every programmer is using them. Code duplication is easy and cheap during software development, but it makes software maintenance more complicated:

- Errors may have been duplicated together with the duplicated code.
- Modifications of the original code often must also be applied to the duplicated code.

Especially for software renovation projects, it is desirable to detect duplicated code; a number of approaches have been developed [3, 2, 5, 7]. These approaches are text-based (and language independent) [3], syntax-based [2] or are based on metrics (syntax- and/or text-based) [5, 7]. Some approaches can only detect (textual or structural) identical duplicates, which are not typical in software systems as most duplicates are adapted to the environment where they are used.

In Figure 1 two similar pieces of code in `main.c` from the `agrep` program are shown, which have been detected as duplicates by our prototype tool. Let us assume that the left part is the original and the right part is the duplicate. We can identify some typical modifications to the duplicate:

1. Parts of the code will be executed under different circumstances (lines 742 and 743 have been moved into an `if` statement in lines 473-476).
2. Variables and/or expressions are changed (lines 743/478, 747/483, ...).
3. Parts of the code are inserted or deleted ("`lasti = i-1`" in line 758).
4. Code is moved ("`j++`" in line 481/748).

Modifications disturb the structure of the code and duplicated code is more complicated to identify. This causes a tradeoff between precision (amount of false positives) and recall (amount of undiscovered duplicates) in text- or structure-based detection methods. To also detect not identical but similar duplicates (increased recall), the methods have to ignore certain properties. However, this may lead to false positives (reduced precision). This tradeoff has been studied in [5].

We have developed an approach which does not suffer under the tradeoff between recall and precision and where modified duplicates can still be detected. Such an approach cannot just be based on text or syntax, but has to consider semantics too. Our approach is based on *fine-grained program dependence graphs (PDGs)* which represent the structure of a program and the data flow within it. In these graphs, we try to identify similar subgraph structures which are stemming from duplicated code. Identified similar subgraphs can be directly mapped back onto the program code and presented to the user.

```
740 if(c != Newline)
741 {
742   r1 = Init1 & r3;
743   r2 = (Next[r3] & CMask) | r1;
744 }
745 else {
746   r1 = Init1 & r3;
747   r2 = Next[r3] & CMask | r1;
748   j++;
749   if(TAIL) r2 = Next[r2] | r2 ;
750   if(( r2 & 1) ^ INVERSE) {
751     if(FILENAMEONLY) {
752       num_of_matched++;
753       printf("%s\n", CurrentFileName);
754       return;
755     }
756     r_output(buffer, i-1, end, j);
757   }
758   lasti = i - 1;
759   r3 = Init0;
760   r2 = (Next[r3] & CMask) | Init0;
761 }
762 c = buffer[i++];
763 CMask = RMask[c];
```

```
472 if(c != Newline)
473 { if(CMask != 0) {
474   r1 = Init1 & r3;
475   r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
476  }
477  else  {
478   r2 = r3 & Init1;
479  }
480 }
481 else { j++;
482   r1 = Init1 & r3;
483   r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
484   if(TAIL) r2 = (Next[r2>>hh] | Next1[r2&LL]) | r2;
485   if(( r2 & 1 ) ^ INVERSE) {
486     if(FILENAMEONLY) {
487       num_of_matched++;
488       printf("%s\n", CurrentFileName);
489       return;
490     }
491     r_output(buffer, i-1, end, j);
492   }
493   r3 = Init0;
494   r2 = (Next[r3>>hh] | Next1[r3&LL]) & CMask | Init0;
495 }
496 c = buffer[i++];
497 CMask = Mask[c];
```

Figure 1: Two similar pieces of code from `agrep`

2 Identifying similar subgraphs

An *attributed directed graph* is a is a 4-tuple $G = (V, E, \mu, \nu)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, $\mu : V \to A_V$ maps vertices to the vertex attributes and $\nu : E \to A_E$ maps edges to the edge attributes. Let $\Delta : E \to A_v \times A_E \times A_v$ be the mapping $\Delta(v_1, v_2) = (\mu(v_1), \nu(v_1, v_2), \mu(v_2))$. A *path* is a finite sequence of edges and vertices $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ where $e_i = (v_{i-1}, v_i)$ for all $1 \leq i < n$. A *k-limited path* is a path $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ with $n \leq k$.

Two attributed directed graphs $G_1 = (V_1, E_1, \mu_1, \nu_1)$ and $G_2 = (V_2, E_2, \mu_2, \nu_2)$ are *isomorphic*, if a bijective mapping $\phi : V_1 \to V_2$ exists with:

$$(v_i, v_j) \in E_1 \iff (\phi(v_i), \phi(v_j)) \in E_2, \quad \Delta_1(v_i, v_j) = \Delta_2(\phi(v_i), \phi(v_j))$$

This means that two graphs are isomorphic if every edge is bijectively matched to an edge in the other graph and the attributes of the edges and the incident vertices are the same. The question *whether two given graphs are isomorphic* is NP-complete in general.

In Figure 2 two simple attributed graphs are shown, where the edge labels represents the complete attribute-tuple of the vertex and edge attributes. At least two maximal isomorphic subgraphs exists with six vertices each. We are interested in *similar* subgraphs which do not have to be isomorphic. We define similarity (which is always tricky) between graphs by relaxing the mapping between edges: We consider two graphs G and G' as similar, if for every path $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ in one graph there exists a path $v'_0, e'_1, v'_1, e'_2, \ldots, e'_n, v'_n$ in the other graph and the attributes of the vertices and the edges are identical if the path are mapped against each other ($\forall_{1 \leq i \leq n} e_i, e'_i : \Delta(e_i) = \Delta'(e'_i)$). The second restriction is that all paths have to start at a single vertex v in G and at v' in G' ($v_0 = v, v'_0 = v'$ for all such paths).

A naive approach to identify the maximal similar subgraphs would now calculate all (cycle free) paths starting at v and v' and would do a pairwise comparison afterwards. This is infeasible: even if the paths are length limited, the length would be unusable small.

Our approach is constructing the maximal similar subgraphs by induction from the starting vertices v and v' and is matching length limited similar paths. What makes this approach feasible, is that it considers all possible matchings *at once*. This is seen at the example:

1. The algorithm starts with $v = 1$ and $v' = 10$. These vertices are considered the endpoints of matching paths of the length zero.

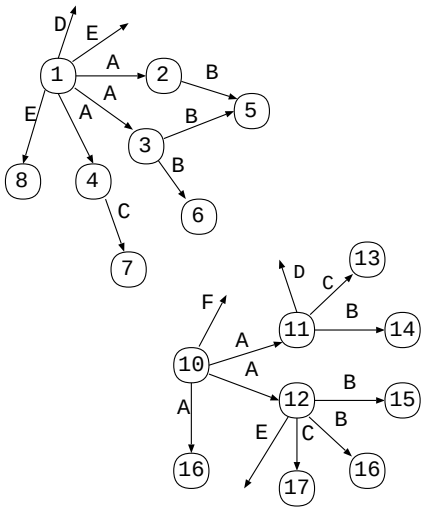


Figure 2: Two simple graphs

```
propagate(V1, V2, l):
If l ≤ k:
  Let V1 ⊂ V and V2 ⊂ V be the the endpoints
    of similar paths.
  Let E1 and E2 be the edges that are leaving
    the vertices of V1 and V2.
  Partition E1 and E2 into equivalence classes
    E1i and E2i based on Δ.
  For all E1i with their corresponding E2i:
    Add edges from E1i and E2i to Gv1 and Gv2
    Let V1i and V2i be the vertices that are
      reached by the edges in E1i and E2i
    Call propagate(V1i, V2i, l + 1)

generate(v1, v2, k):
Call propagate({v1}, {v2}, 1)
Return Gv1 and Gv2 as result.
```

Figure 3: Algorithm to generate G_{v_1} and G_{v_2}

2. Now, the matching paths are extended: The incident edges are partitioned into equivalence classes based on the attributes. There is only one pair of equivalence classes that share the same attributes in both graphs: $\{(1,2),(1,3),(1,4)\}_A$ and $\{(10,11),(10,12),(10,16)\}_A$.

3. The reached vertices are now marked as being part of the maximal similar subgraphs and the algorithm is continuing with the sets of reached vertices $\{2,3,4\}$ and $\{11,12,16\}$.

4. Again the incident edges are partitioned into the first pair $\{(2,5),(3,5),(3,6)\}_B$ and $\{(11,14),(12,15),(12,16)\}_B$ and the second pair $\{(4,7)\}_C$ and $\{(11,13),(12,17)\}_C$. For both pairs the algorithm continues recursively.

5. The reached vertices $\{5,6\}$ and $\{14,15,16\}$ are marked as parts of the maximal similar subgraphs. No edges are leaving these vertices.

6. The other set pair of reached vertices $\{7\}$ and $\{13,17\}$ are marked. No edges are leaving.

7. No more set pairs exists, the algorithm terminates.

In the end, the algorithm has marked $\{1,2,3,4,5,6,7\}$ and $\{10,11,12,13,14,15,16,17\}$ which induce the maximal similar subgraphs.

A simplified version of the algorithm is is shown in Figure 3. It calculates the maximal similar subgraphs G_1 and G_2 which are induced by k-limited paths starting at the vertices v_1 in G_1 and v_2 in G_2. We call these graphs *maximal similar k-limited path induced subgraphs* $G^k_{v_1}$ *and* $G^k_{v_2}$.

Before maximal similar k-limited path induced subgraphs G^k_v and $G^k_{v'}$ can be found, the possible pairs (v,v') have to be detected. A naive approach would be to check all pairs $V \times V$ which leads to a complexity of $O(|V|^2)$ (independent of the complexity of the generation of the subgraphs them self). Even with smarter approaches, this complexity cannot be reduced. Therefore, only a subset of V should be considered as "starting" vertices, as other vertices are probably reached during the construction of the maximal subgraphs.

3 Implementation

The presented technique has been implemented in a prototype on top of our infrastructure to analyze ANSI-C programs [6]. This infrastructure represents analyzed programs in *program dependence graph (PDG)* [4], which are directed attributed graphs whose vertices represent the assignment statements and control predicates that occur in a program. Some of the vertices have an attribute that marks them as entry vertices, which represent the entry of functions. The edges represent the *dependences* between the components of the program. They have two attributes: the first is separating the edges into *control* and *data dependence edges* and

the second is `true` or `false` for control dependence edges. A control dependence edge from vertex v_1 to v_2 represents that if the predicate that is represented by v_1 is evaluated to the second attribute of the edge, the component that is represented by v_2 will be executed. A data dependence edge from vertex v_1 to v_2 represents that the component represented by v_1 assigns a value to a variable which may be used at the component represented by v_2.

3.1 Fine-grained PDGs

Our *fine-grained* PDG is a specialization of the traditional and is similar to both the AST and the traditional PDG. On the level of statements and expressions, the AST vertices are almost mapped one to one onto PDG vertices. The definitions of variables and functions have special vertices. The vertices may be attributed with a class, an operator and a value. The class specifies the kind of vertex: statement, expression, function call etc. The operator further specifies the kind, e.g. binary expression, constant etc. The value carries the exact operator, like "+" or "–", constant values or identifier names. Between vertices that represents components of expressions we have also specialized edges, which are attributed with their class (control, data, etc.) and label.

To find similar code based on identifying maximal similar subgraphs in fine-grained PDGs we first had to find the subset of the vertices which are used in the pairwise construction of the subgraphs. One possibility would have been to use entry vertices, which would find similar functions. We decided to use predicate vertices instead, because we also want to find similar pieces of code independent of functions. For every pair of predicate vertices (v_1, v_2) the maximal similar $G^k_{v_1}$ and $G^k_{v_2}$ are generated. The generation is basically a recursive implementation of the induction from Figure 3.

3.2 Weighted subgraphs

If we take the subgraphs as direct result, they just represent *structural* similarity which can also be achieved via less expensive techniques like [2]. The subgraphs can be large even if they do not even have a similar semantic. The reason is that the data dependence edges may not match and the subgraphs are only or mostly induced by control dependence edges. Only if the data dependence edges are considered special it is guaranteed that the subgraphs have a similar semantic. Therefore the constructed subgraphs have to be weighted. A simple criterion is just the number of data dependence edges in the subgraphs. As our evaluation in the next section shows, this criterion is good enough. However, other, more sophisticated criterions are possible like the percentage of data dependence edges or the amount and the length of paths induced by data, value and reference dependence edges.

4 Evaluation

Like any other k-limited technique, the presented work had to be "tuned" to find an appropriate value for k. We therefore checked a set of test programs stemming from different sources for duplicated code. The results can be seen for some examples in Figure 4. The size of the programs are given in terms of lines of code and the number of vertices and edges in the PDG. For different limits k between 10 and 50 the running times are given (measured in seconds of user time spent). A direct relation between the size of a program and the running time does not exists as the running time is mostly dependent on the size and the amount of similar subgraphs within a program. However, due to the pairwise comparison we expect a quadratic complexity overall. In the same table, the last three columns show the amount of discovered duplicates with a minimum weight of 10, 20 and 50. The limit used was $k = 20$ and only minimal differences exists for larger k (except for `twmc`). Due to lack of time it was impossible to manually verify all reported duplicates. However, all reported duplicates we checked were correct (100% precision).

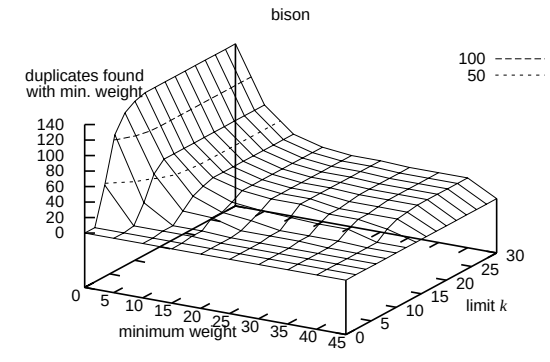
Due to the complexity of the data flow analysis used in our infrastructure, we are only able to construct PDGs up to a limited size of programs. This does not mean that the presented technique has the same limit—we need a reimplementation on top of a different infrastructure to fully evaluate for big programs.

4.1 Optimal limit

To insure highest possible recall, a very high k-limit is desirable. However, this is not possible due to the exponential complexity of the graph comparison. Our claim is that a small k is sufficient and that a limit above this small value will not increase recall. We found this claim to be true for almost any test case. A typical case is `bison`, for which the results are shown in Figure 5. All test cases were repeated for limits $0 \leq k \leq 30$ (y-axis). Also shown is how many duplicates (z-axis) are

Project	LOC	Edges	Vertices	Time f. limit k (sec)					Duplicates		
				k=10	k=20	k=30	k=40	k=50	$\geq$10	$\geq$20	$\geq$50
`agrep`	3968	69032	22588	43.6	368.4	2662.9	-	-	155	95	14
`bison`	8303	79030	28071	15.9	73.6	382.7	1111.1	1446.68	36	22	0
`cdecl`	3879	40578	12939	0.8	0.9	0.9	0.9	0.9	0	0	0
`compiler`	2402	99219	16497	467.6	482.3	482.4	482.4	482.4	95	57	40
`diff`	17485	169508	43518	3.9	13.2	53.0	126.1	141.2	40	10	6
`fft`	3242	35701	16446	12.3	77.7	388.1	1272.4	1899.78	16	12	3
`flex`	7640	124730	37073	5.0	6.0	6.5	6.6	6.5	16	0	0
`football`	2261	63833	18718	55.4	104.6	111.8	111.9	112.0	50	2	0
`larn`	10410	817432	158077	612.4	12883	15321.4	-	-	107	59	6
`patch`	7998	196106	29766	8.2	9.7	11.7	12.6	12.6	2	0	0
`rolo`	5717	50816	17438	1.1	1.1	1.1	1.1	1.1	0	0	0
`spim`	19739	1338294	122819	768.5	991.1	1108.7	1107.53	1125.68	40	23	6
`twmc`	24950	1605532	181281	1387.9	37387	-	-	-	1417	785	380

Figure 4: Sizes and running times[1] for some test cases



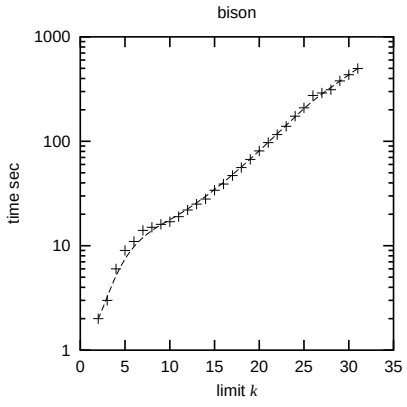


Figure 5: Results[1] for `bison`

reported that are above a specific minimum weight (y-axis). As we can see, for very small k ($< 5-10$) almost no duplicates are reported. For bigger (but still small) k ($< 15-20$) the amount of reported duplicates is increasing fast. For bigger k (> 20) the amount of reported duplicates is not changing any more. We have found this to be the same for almost any other test case—a k-limit around 20 seems to be sufficient for highest recall.

4.2 Minimum weight

The other "tunable" parameter in our technique is the minimum weight of a similar subgraph before it is reported. This value is not critical like the k-limit, as it does not influence the comparison itself. Normally, all possible duplicates are identified independent of their weights and the minimum weight just changes the amount of *reported* duplicates. The `bison` test case is an ideal example: for small minimum weights, many duplicates are reported. For bigger minimum weights this changes quickly, which shows that the majority of duplicates are small pieces of codes. For minimum weights between 10 and 40, around 40 duplicates are reported. For minimum weights above 45, no duplicates are reported, which shows that the maximum weight of all duplicates is less than 45.

We have found that there is no "ideal" minimum weight, as every test case has different amounts of re-

[1] Variations in time results are due to parallel running tasks.

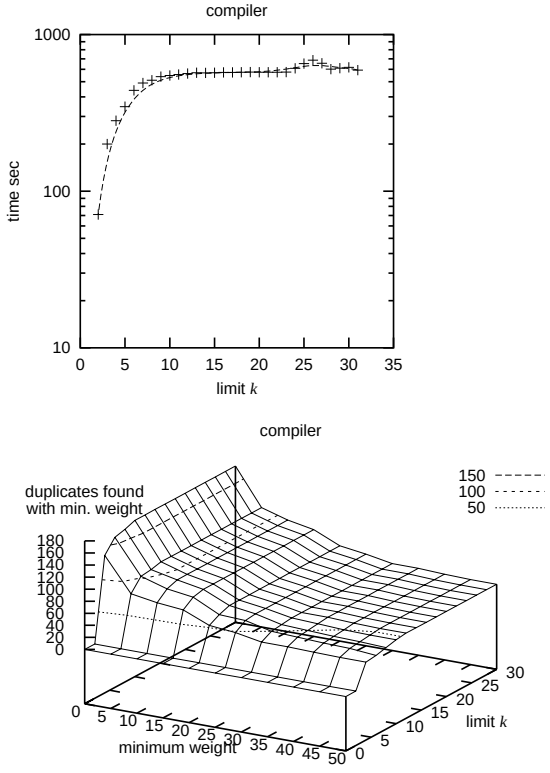


Figure 6: Results[1] for `compiler`

ported duplicates with varying minimum weights. This is not unexpected, as duplication is different in every program.

4.3 Running time

Figure 5 also shows the times for the `bison` example, which are increasing exponential for large k. We claimed that a k-limit around 20 is ideal for recall: we need 73 seconds to analyze `bison` under this limit. For some test cases we have found an interesting behavior—the running time is not increasing exponential but reverse logarithmic for increased k. This is shown in Figure 6 for the test case `compiler`. As you can see, for k-limits bigger than ten the amount of reported duplicates stays the same: there are more than 50 duplicates with a weight bigger than 50. This means that there are no similar paths longer than 10 edges in that software and the limit is not reached for larger limits: The time needed to calculate the similar graphs is independent of k for k bigger than 10. Therefore, the

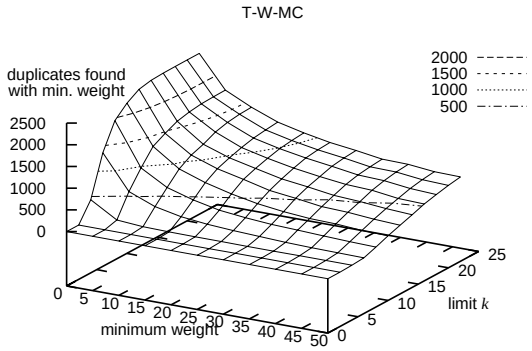


Figure 7: Results for `twmc`

overall needed time is not changing above that. The same behavior can be seen for the test cases of Figure 4 with few duplicates: half of the test cases have no big difference in running time for the limits $k = 40$ and $k = 50$.

One of our test case (see Figure 7) was different than all others: First of all, we could not test for k-limits bigger than 22, as the running time was already at 26 hours. Also, the amount of reported duplicates was incredibly high: more than 500 with a weight bigger than 50 and more than 1000 with a weight bigger than 20. These extreme high numbers are stemming from massive code duplication in that particular software. We have found a high amount of files, which just have been copied and slightly changed for slightly different purpose.

5 Related Work

To our knowledge, our approach is the first that obeys the data flow in a program and not only the (syntactical) structure. Nearest to our work is probably [2], where a program under observation is transformed to an AST, for every subtree in the AST a hash value is computed and identical subtrees are identified via identical hash values. To also detect similar not identical subtrees, the subtrees have to be pairwise compared. The authors suggest many improvements as future work which are similar to our approach.

Another approach which obeys syntactical structure is [7], where metrics are calculated from names, layout, expression and (simple) control flow of functions. Two functions are considered as clones if their metrics are similar. This work can only identify similar functions

but not similar pieces of code. A language independent approach is [3] which is looking for specific patterns in a comparison from every line to every other. Another text-based approaches is [1].

An application in the same setting is the detection of *plagiarism*: Given two programs, one has to detect if one program is in part or completely duplicated in the other. Most plagiarism ldetecting systems like [8] are comparing the lexical structure of the programs. Other system are again based on metrics; however, studies show that metrics-based systems are only partly successful because of the tradeoff between recall and precision, both for detection of plagiarism [9] and detection of similar code [5].

6 Summary and future work

We have presented a technique for identifying similar code based on finding maximal similar subgraphs in fine-grained program dependence graphs. As this problem is not solvable in polynomial time, a *k*-limiting technique is used. A prototype implementation shows that this approach is feasible even with the non polynomial complexity of the problem and results in high precision and recall.

This is work in progress and some obstacles remain to be solved: First off all, high amounts of duplicated code cause exploding running times. Secondly, large duplicated code sections cause many duplicates to be reported, as duplicates are basically reported for every predicate within. These duplicates are overlapping and have to be merged before reported to the user.

Our future plans include:

- A reimplementation on top of a simpler infrastructure to enable an evaluation for large programs. Due to the underlying infrastructure, our prototype is only able to analyze programs up to limited size.

- An adaption of our prototype for detection of plagiarism. We are using JPlag [8] in education with great success. However, a manual check is still needed as students are aware of our tool usage and try to hinder the detection through simple modifications. A plagiarism detection tool based on our approach should not be so easily confused.

- An automatic substitution of identified duplicated code through new functions or macros. As the underlying infrastructure contains enough semantic information in the PDGs, the *isomorphic* subgraphs can be identified and replaced by new parameterized function calls which do not change the semantic of the program.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, 1995.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, 1998.

[3] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *IEEE International Conference on Software Maintenance*, 1999.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[5] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Fourth Working Conference on Reverse Engineering*, 1997.

[6] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12), 1998.

[7] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, 1996.

[8] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[9] K. L. Verco and M. J. Wise. Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, 39(9), 1996.