Detecting Refactorable Clones by Slicing Program Dependence Graphs

Ammar Hamid[1] and Vadim Zaytsev[2]

[1] ammarhamid84@gmail.com
[2] vadim@grammarware.net
Institute of Informatics, University of Amsterdam, The Netherlands

Abstract. Code duplication in a program can make understanding and maintenance difficult. The problem can be reduced by detecting duplicated code, refactoring it into a separate procedure, and replacing all the clones by appropriate calls to the new procedure. In this paper, we report on a confirmatory replication of a tool that was used to detect such refactorable clones based on program dependence graphs and program slicing.

1 Motivation

With the discussion about the extent to which code clones are harmful for software readability, maintainability and ultimately quality, still ongoing, there is still significant evidence on cost increases being caused by code duplication in at least some scenarios [5,18]. For simplicity, we intend to adopt that view and look a step further. Once clones are identified, ideally we would like to provide advanced support for programmers or maintenance engineers to remove them — that is, to use refactorings [4] to "de-clone" source code by merging identical code fragments and parametrising similar ones [17].

The sheer number of code clone detection techniques and tools is immensely overwhelming [15,16,13]. In section 2, we will give a very brief overview of the field and explain terminology needed to understand the rest of the paper. One of the promising family of methods which is not too complex for a final Master's project yet also not too much of a beaten track in code clone research, is *graph-based*. Given two programs, we automatically build a graph-like structure with known properties, employ some slicing and/or matching and based on that can diagnose them with duplication.

Eventually we have converged to a relatively well-known paper of Raghavan Komondoor and Susan Horwitz [8] and dedicated ourselves to replicating it. Some details about that project can be found in section 3, but in general they propose to use program dependence graphs [11] (PDG) and program slicing [19]. The authors of the original study were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique using a combination of backward slicing and forward slicing. Basically they search for sets of syntactically equivalent node pairs and perform backward slicing from each pair with a

single forward slicing pass for matching predicates nodes. The theoretical foundation behind this method mostly lies in plan calculus [14] and an advanced graph partitioning algorithm [6,7] and essentially allows to detect clones semantically, regardless of various refactorings that may have been applied to some of the copies but not to others. This leads to reporting only those clones that can indeed be refactored — as we show on Table 1.

We modified the original study in several ways: some were forced upon us by technicalities, for others we had our own reasons — all explained in section 4. We report our results, compare them to the original study and try to explain the differences in section 5 and conclude the paper with section 6.

2 Background

Several studies show that 7–23% of the source code for large programs is duplicated code [2,9]. Within a project, code duplication will increase code size, and make maintenance difficult. Fixing a bug within a project with a lot of duplicated code is becoming a challenge because it is necessary to make sure that the fix is applied to all of the duplicated instances. Lague et al [10] studied the development of a large software system over multiple releases and found that programmers often missed some copies of the duplicated code when performing modification. Similar results have been observed by Geiger et al [5] and Thummalapenta et al [18] who observe the already expected negative impact of clone co-evolution on software maintenance effort.

In code duplication studies we usually distinguish among the following types of clones [13,15]:

- *Exact clones* (type 1) — identical duplicates with some variations allowed in whitespace and comments;
- *Parametrised clones* (type 2) — variations are allowed in identifier names, literals, even variable types;
- *Near miss clones* (type 3) — statements are allowed to be changed, added or removed up to some extent;
- *Semantic clones* (type 4) — same computation with a different syntax and possibly even different algorithms;
- *Structural clones* — higher level similarities, conceptually bottom-up-detected implementation patterns;
- *Artefact clones* — function clones and file clones;
- *Model clones* — duplicates over artefacts other than code;
- *Contextual clones* — code fragments deemed duplicate due to their usage patterns.

Out of these, type 2 and type 3 are the most well-researched ones, with model clones quickly getting more and more attention every year.

Techniques and tools can be roughly classified into these groups [13,16] (in the parenthesis we show a software artefact category in the terms of parsing-in-a-broad-sense megamodel [20]):

Procedure 1

```
 int foo(void) {
++   int i = 1;
     bool z = true;
     int t = 10;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Rewritten Procedure 1

```
 int foo(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Procedure 2

```
 int bar(void) {
++   int i = 1;
     bool w = false;
     int t = 10;
++   int s;
++   int b = a + 1;
++   for (s=0; s<10; s++) {
++      b = b + 5;
     }
++   int c = a + b - 1;
     return c;
 }
```

Rewritten Procedure 2

```
 int bar(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Newly extracted procedure:

```
 int new_procedure_bar(void) {
++   int i = 1;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Table 1. Two functions with duplicated code and a refactoring result. In the left column, the duplicated code is marked with `++`; in the right column clones are replaced with calls to a newly extracted function. This example demonstrates that not everything that has the same structure or the same syntax is reported as clones (e.g. `int t = 10;` which has no shared predecessor).

– *Text based* (Str) such as SIMIAN — blazingly fast methods usually looking for exact clones, quite often in a language-independent, -parametric or -agnostic manner;
– *Token based* (TTk, Lex) such as CCFINDER — somewhat more sophisticated lexical tools;
– *Tree based* (Ptr, Cst, Ast) such as DECKARD — looking for clones in parse trees, suffix trees or abstract syntax trees;
– *Graph based* (Ast, Dia) such as DUPLIX — making decisions based on control flow graphs, data dependency graphs, program dependence graphs or partite sets and vertices;
– *Model based* (Dia) such as CONQAT — metamodel-specific representation, usually graph-like;
– *Metrics based* such as COVET — using metrics, fingerprinting and/or clustering to work on text or ASTs;
– *Hybrid* such as CLONEMINER — independent component analysis, some variants of semantic indexing and longest subsequence methods that require reasoning over trees, memory states, vector spaces, etc.

Following the original paper [8], we use CodeSurfer[3], a commercial tool that can be used to generate program dependence graphs (PDGs) from C programs. It provides an API that can be used from Scheme programs [1]. In general, PDG nodes represent program statements and predicates, while PDG edges represent data and control dependencies. PDG provides an abstraction that ignores arbitrary sequencing choices made by a programmer, and instead captures the important dependences among program components. Essentially, a program dependence graph is built starting from a control flow graph (CFG) with statements as nodes and possible transitions among them as edges, which is then analysed for dominance to form an acyclic post-dominator graph — the two are merged into a control dependence graph. A program dependence graph is formed from the control dependence graph by enhancing it with additional edges for all data dependencies, an example is given on Figure 1. The resulting complex structure is graph-like with nodes of several kinds (regions, statements, entry/exit points) and edges of several kinds (data/control dominance, possibly labelled) — there are algorithmic variations which are not important for understanding the current paper. Such a PDG is remarkable in a sense that it captures many structural aspects of a program and still allows to abstract from concrete details such as variable names and precise positioning of the code. For a larger/smaller scale, related methods are used such as system dependence graphs (SDGs) or execution dependence graphs (EDGs) [12].

The last bit of background needed for understanding this paper is program slicing [19,3], which is a well-known technique for obtaining a "view" of a program with only those statements that are relevant for the chosen variable. In terms of PDG we can have two query types in program slicing [7]. *Backward slicing* from node x means finding all the nodes that influence the value of node

[3] CodeSurfer, http://www.grammatech.com/research/technologies/codesurfer.

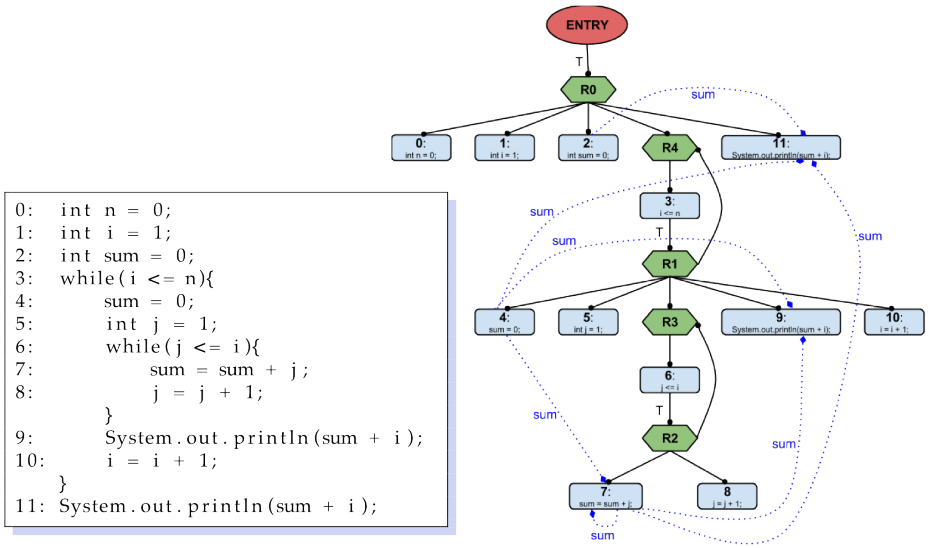

Fig. 1. A tiny code fragment demonstrating the concept of a program dependence graph: the listing on the left; the corresponding PDG fragment on the right (only data dependencies for *sum* are shown, the complete graph is much bigger and more cluttered) [21].

x. *Forward slicing* from node y means finding all the nodes that are influenced by node y. This is an important technique to filter out any statements that are irrelevant for clone detection.

3 Original study

The main research question asked by Komondoor and Horwitz is the following: can we find code clones of type 3 (non-contiguous, reordered, intertwined), which are refactorable into new procedures? [8]

3.1 Approach

To detect clones in a program, we represent each procedure using its PDG. In PDG, vertex represents program statement or predicate, and edge represents data or control dependences. The algorithm performs four steps (described in the following subsections):

- *Step 1:* Find relevant procedures
- *Step 2:* Find pair of vertices with equivalent syntactic structure
- *Step 3:* Find clones
- *Step 4:* Group clones

Find relevant procedures. We are only interested in finding clones for procedures that are reachable from the main program execution. Only then we can safely remove unreachable procedures from our program and just not detect clones of them. We do this by getting a system initialisation vertex and forward-slicing with data and control flow. This will return all PDGs (including user defined and system PDGs) that are reachable from the main program execution. From that result, we further filter those PDGs to find only the user defined ones, ignoring system libraries.

Find pairs of vertices with equivalent syntactic structure. We scan all PDGs from the previous step to find vertices that have the type *expression* (e.g. `int a = b + 1`). From those expression vertices, we try to match their syntactic structure with each other. To find two expressions with equivalent syntactic structures, we make use of Abstract Syntax Tree (AST). This way, we ignore variable names, literal values, and focus only on the structure, e.g. `int a = b + 1` is equivalent with `int k = l + 1`, where both expression has the same type, which is *int*).

Find clones. From a pair of equivalent structure expressions, we back-slice to find their predecessors and compare them with each other. If the AST structures of their predecessors are the same then we store it in the collection of clones found. Because of this step, we can find non-contiguous, reordered, intertwined and refactorable clones. Refactorable clones in this case mean that the found clones are meaningful and it should be possible to move it into a new procedure without changing their semantic.

Group clones. This is the step where we make sense of the found clones before displaying them. For example, when using CodeSurfer, the vertex for a while-loop doesn't really show that it is a while loop but rather showing its predicate, e.g. `while(i<10)` will show as a control-point vertex `i<10`. Therefore, it is important that the found clones are mapped back to the actual program text representation and grouped together before displaying them. It is important that the programmer can understand and take action on the reported clones.

Experimental setup. The authors of the original paper used CodeSurfer version **1.8** to generate PDGs and wrote Scheme program of **6123** lines that access CodeSurfer API to work with the generated PDGs. They also had to have a C implementation of **4380** lines to do the processing of those PDG to actually find clones.

They were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique that used a combination of backward slicing and forward slicing. They applied it to some open-source software written in C (`tail`, `sort`, `bison`) and demonstrated the capability of slicing to detect non-contagious code clones. We will show the actual numbers later when we compare the results with the replication.

4 Changes to the original study

The motivation of this replication study is to be able to validate algorithm and results of the original study. Once validated, we would like to publish our code and intermediate results into a public repository so that it is easier for any future researchers to either re-validate our results or to extend our program.

We had to use CodeSurfer 2.3 instead of 1.8 used in the original study: just to get it running was already a challenge impossible to overcome — we would eventually need to do a sandboxing of some 2001 version of OS, which would then need to be properly licensed (CodeSurfer is not open source, but we have applied for the academic license and got both 1.8 and 2.3 to experiment with).

Porting the existing code (kindly provided to use by Raghavan Komondoor) to the new version of CodeSurfer was also ruled out as a viable option: the API changed too much, and actually covered many things with standard calls that needed to be programmed in full when working with version 1.8. In the end, we reimplemented the algorithm from scratch, using both the original paper and the code behind it as guides. Our implementation has **536** LOC of Scheme, which is huge improvement against the 6123 LOC of the original study. The improvement is mostly not ours to claim, but CodeSurfer API's. For post-processing of clones, we wrote a Ruby script, which was again shorter: 161 LOC versus the original 4380 LOC, partly due to the improved API, but partly also due to the language choice (the original post-processing was done in C++). Actually, given a bit more time, it should have been possible to avoid post-processing entirely, or rather to implement in all in Scheme. The code is available online for anybody to do this — `http://github.com/ammarhamid/clone-detection` — we accept pull requests.

There are several other important changes from the original paper that we need to explain. As mentioned above, we only detect clones within the reachable procedures, excluding any unused procedures that are not reachable from main execution. This makes the result more accurate, since dead code is out of our consideration.

Furthermore, we only use backward slicing and no forward slicing to detect clones. Let us have a look at the example on Table 2. According to the original paper, only statements indicated by `++` will be reported as clones while statement marked with `**` is excluded. The main argument according to the original paper is that `fp3` is used inside a loop but the loop predicate itself is not matching (for loop and the first while loop predicate doesn't match) – or a so called cross loop [8].

However, we argue that we should still report the statement marked with `**` as a clone together with the fact that their loop predicate doesn't match. For a software developer it would mean one could still refactor this into two separate procedures, instead of a single procedure proposed by the original paper (Table 3 and Table 4). Therefore, we consider that *forward slicing is only necessary to define refactoring strategy* and not for *detecting the clone* itself.

Fragment 1	**Fragment 2**

`...`
`** fp3 = lookaheadset + tokensetsize;`
`for(i = lookaheads(state);`
`i < k; i++) {`
`++ fp1 = LA + i * tokensetsize;`
`++ fp2 = lookaheadset;`
`++ while (fp2 < fp3) {`
`++ *fp2++ |= *fp1++;`
`}`
`}`
	`...`
	`** fp3 = base + tokensetsize;`
	`...`
	`if(rp) {`
	`while((j = *rp++) > 0) {`
	`...`
	`++ fp1 = base;`
	`++ fp2 = F + j * tokensetsize;`
	`++ while(fp1 < fp3) {`
	`++ *fp1++ |= *fp2++;`
	`}`
	`}`

Table 2. Two clones from bison that illustrates the necessity to have a forward slicing according to the original paper [8]

The new fragment 1	**The new fragment 2**

`...`
`fp3 = location(lookaheadset,`
`tokensetsize);`
`...`
`for(i = lookaheads(state);`
`i < k; i++) {`
`compute(LA, lookaheadset,`
`i, tokensetsize, fp3);`
`}`
	`...`
	`fp3 = location(base, tokensetsize);`
	`...`
	`if(rp) {`
	`while((j = *rp++) > 0) {`
	`...`
	`compute(F, base,`
	`j, tokensetsize, fp3);`
	`}`
	`}`

Table 3. The new fragments after refactoring (without forward slicing)

The extracted procedures

```
int location(int base, int size) { return base + size; }

void compute(int cons, int base, int index, int size, int loc) {
    fp1 = cons + index * size;
    fp2 = base;
    while (fp2 < loc) { *fp2++ |= *fp1++; } }
```

Table 4. The new refactored procedures. In this case, procedure location has only one statement which probably unnecessary to create a new procedure for it. But the point is if we use forward slicing in clone detection phase, we might hide this statement prematurely from the programmers, who at least should be aware of the situation before proceeding with refactoring.

Study	Program	LOC	PDG nodes	Elapsed time, minutes:seconds		
				Scheme	C++	Ruby
Original	tail	1569	2580	00:40	00:03	—
Replication	tail	1668	3052	00:05	—	00:01
Original	sort	2445	5820	10:00	00:07	—
Replication	sort	2499	6891	00:30	—	00:01
Original	bison	11540	28548	93:00	01:05	—
Replication	bison	10550	33820	126:00	—	00:42

Table 5. Comparison on program size, number of nodes, implementation and time.

5 Results

To be as close to the original paper as possible, we used the GNU git repositories[4] to locate versions that were released around 2001: CoreUtils 4.5.2 (for `tail` and `sort`) and Bison 1.29 (for `bison`).

Table 5 shows the comparison of the sizes of the three programs (in number of LOC and in number of nodes), and the running times for the algorithm between the original and replication study. Figure 2 shows the comparison of the result in details between the original and replication study.

We do not have a solid explanation for the differences observed, but we can hypothesise on some issues:

Altered algorithm. We did use a slightly different algorithm (only reachable code; no forward slicing) to detect clones. However, we have also tried running it exactly as it was intended originally, and the differences were rather minor and could not explain some of the drastic differences.

Manual inspection was performed to ensure that the clones detected by our tool are indeed clones and are indeed refactorable. It was possible to review all clones from `tail` and `sort` and cover a random selection of clones for `bison` — no false positives were found.

Bison running time in the original study is suspiciously short, which does not reflect the explosive performance behaviour that we have observed in our implementation. This could indicate a bug in one of the implementations, or point to a drastically different (optimised, distributed) algorithm used for the actual run of the original experiment. It could also be a simple reporting mistake (e.g., "one and a half hours" reported instead of actual "one and a half days").

Size of some clones reported for `tail` and `bison` is longer than most functions (group 70+), which means either a mistake or some unreported procedure used in the original experiment to combine several subsequent full-function clones into one.

Testing a program of 10 KLOC is always harder than testing a program of 1 KLOC, especially if both programs are algorithmically heavy yet the shorter one relies on a more advanced API. More investigation is needed to see which of these factors were at play and which results are closer to the truth.

[4] http://git.savannah.gnu.org/cgit/

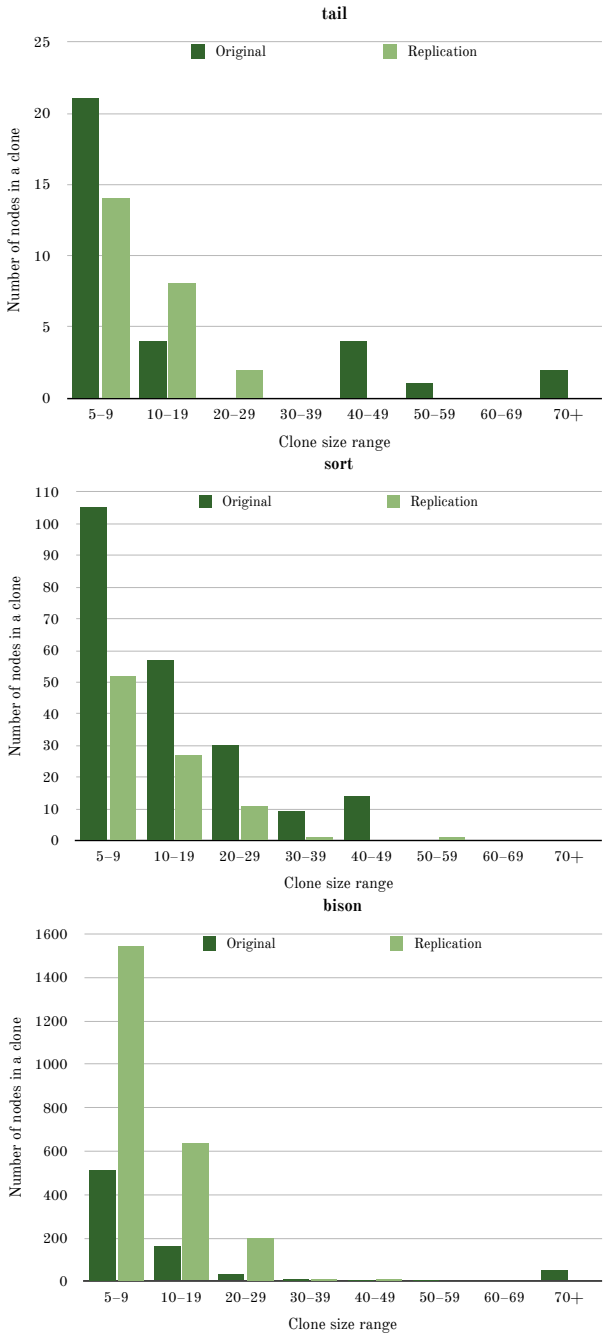

Fig. 2. Detailed comparison results between the original and replication study

6 Conclusion

We have departed on a quest to find refactorable semantic clones and have conducted a replication of a paper that did it with PDG and program slicing. Our results are statistically somewhat different from the results of the original study, but we can conclude nevertheless that the algorithm described there, works. So, *the fusion of PDG and slicing is suitable for Type 3 clone detection.*

As a side product, we have noticed how significantly CodeSurfer has improved over the years: the amount of code we needed to write to achieve the same objectives, is ten times less than what had to be done 13 years ago, with almost no postprocessing of the obtained results needed.

As for quantitative differences, unfortunately we could not compare them in detail since we lack the original data, and we failed in getting the code operational (it would require an old version of CodeSurfer operating on an old system, preferably with performance comparable to the machine used for the original experiment). However, we do present some evidence of correctness in the form of manually reviewed code clones that we reported. We can also conclude that the clones are indeed refactorable — this has been evaluated through manual inspection of the tool reports.

Both the code and the intermediate results of our experiments have been shared as open source: `http://github.com/ammarhamid/clone-detection`, to make it easier to revalidate, replicate, and extend. We hope our clone detector is a suitable tool to use for future work. Possible future extensions should include detecting interprocedural clones as well, which would allow detection of type 4 clones and refactorings such as inlining variables and extracting methods. Intuitively, it would be more useful to provide results over bigger related code fragments — however, the practical consequences remain to be seen.

Acknowledgement

We would like to thank Raghavan Komondoor for sharing his code: we ended up not using it directly, but having it at hand helped us to understand some details and make a better comparison. We would also like to thank David Vitek from GrammaTech for helping us obtaining an academic license for CodeSurfer and trying to work us through the changes from 1.8 to 2.3 — again, we ended up not opting for a migration, but quickly estimating that to be too much of an undertaking, was a part of this project's success. Last but not least, our thanks go to the participants of SATToSE 2014 for all the discussions we have had in L'Aquila.

References

1. H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, I. Adams, N. I., D. P. Friedman, E. Kohlbecker, J. Steele, G. L., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

2. B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pages 86–95, 1995.
3. J. Beck and D. Eichmann. Program and Interface Slicing for Reverse Engineering. In *ICSE*, pages 509–518. IEEE, 1993.
4. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design or Existing Code*. Addison-Wesley Professional, 1999.
5. R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of Code Clones and Change Couplings. In *FASE*, pages 411–425. Springer, 2006.
6. S. Horwitz. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In B. N. Fischer, editor, *PLDI*, pages 234–245, 1990.
7. S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM ToPLaS*, 12(1):26–60, 1990.
8. R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS*, pages 40–56. Springer, 2001.
9. K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *ASE*, 3(1/2):77–108, 1996.
10. B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *ICSM*, pages 314–321, 1997.
11. K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *SDE*, pages 177–184, 1984.
12. J. Qiu, X. Su, and P. Ma. Library Functions Identification in Binary Code by Using Graph Isomorphism Testings. In Y.-G. Guéhéneuc, B. Adams, and A. Serebrenik, editors, *SANER*, pages 261–270. IEEE, Mar. 2015.
13. D. Rattan, R. K. Bhatia, and M. Singh. Software Clone Detection: A Systematic Review. *Information & Software Technology*, 55(7):1165–1199, 2013.
14. C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM, 1990.
15. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *SCP*, 74(7):470–495, 2009.
16. C. K. Roy, M. F. Zibran, and R. Koschke. The Vision of Software Clone Management: Past, Present and Future. In S. Demeyer, D. Binkley, and F. Ricca, editors, *CSMR-WCRE*, pages 18–33. IEEE, 2014.
17. R. Tairas. Clone Detection and Refactoring. In *OOPSLA*, pages 780–781, 2006.
18. S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An Empirical Study on the Maintenance of Source Code Clones. *EMSE*, 15(1):1–34, 2010.
19. M. Weiser. Program Slicing. *IEEE TSE*, 10(4):352–357, 1984.
20. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.
21. L. Zhang. Implementing a PDG Library in Rascal. Master's thesis, Universiteit van Amsterdam, The Netherlands, Sept. 2014.