A short overview on Program slicing in Software Maintenance

Markus Wolf, Boris Dabrowski, and Harald Steinlechner

Software Maintenance and Evolution VU, TU Vienna

Abstract. Maintainers of large software products often have a challenging job when it comes to understanding complex parts of the software. Additionally the effects of software aging make the job even harder. Program slicing, that is automatically separating different aspects of large programs, therefore greatly supports software maintenance. In this paper we give a short overview on applications for program slicing focusing on the traditional software development phases and its implementation techniques covering static slicing, dynamic slicing and various other slicing techniques.

1 Introduction

Program slicing, which in essence is a reverse engineering technique, has been widely studied since it was first introduced by Weiser in [1] as a decomposition technique that extracts program statements relevant to a specific computation. In this respect a program slice contains all the parts of a progam that may affect the values of some point of interest, the so called slicing criterion. The process of finding such program slices for a given slicing criterion is therefore referred to as program slicing.

There are basically two types of program slices. Static program slices, originally introduced by Weiser, preserve the behaviour of a program with respect to all possible program inputs whereas dynamic program slices only preserve the behaviour for a specific input.

The various applications of Program slicing are caused by two properties of themselves with the additional fact that such slices are independent programs that faithfully represent a subset of the original code. The first property being that in case of program modification only the relevant parts need to be comprehended rather than the whole program. This leads to the application of slicing in maintenance, testing and debugging such as described in [2] and [3]. In order to comprehend a whole program it is easier to start off by comprehending smaller sections and their relations instead of trying to comprehend the whole program directly. Applications such as program comprehension [4], program paralellization and measurement result from this second property. With respect to program comprehension static slicing is helpful to gain a general understanding of those statements that result in the selected computation. Although it has it's advantages the resulting program slices often are still large subprograms because of their imprecise computation. Additionally static slices cannot be used in the process of understanding program execution. On the other hand, dynamic slices often are much smaller than static slices and moreover can be used to understand program execution as proposed in [5].

Therefore program slicing can be seen as a handy tool when it comes to improving the whole software development process and maximise cost efficiency [6].

The remainder of this paper is structured as followed. After a short description of related research in chapter 2 an overview over basic slicing techniques is given in chapter 3. Chapter 4 then focuses on the theoretical and implementational background of slicing and chapter 5 describes the incorporation of slicing in software development phases. Chapter 6 gives a short overview of tools and implementations and this paper then concludes with chapter 7.

2 Related work

There have been many surveys on program slicing in the past such as [7] and [8] but many of them only reviewed parts of researches on program slicing or have been out of date. [9] on the other hand presents a more up to date overview about most of existing program slicing techniques including static slicing, dynamic slicing and the latest more underpin techniques. It discusses contribution of each work and compares the major differences between them.

Three major approaches to program slicing have been proposed since Weisers original concept based on iteration of data flow equations in [10]. His approach works with the control flow graph, computing directly relevant statements for each node in the graph and than gradually adding indirectly relevant statements until no more relevant statements are found. The problem of finding the smallest possible program slice for a given criterion cannot be solved but research in this area resulted in good approximations using data flow equations or graph representations [11].

Another approach results in using information-flow relations to compute program slices. Slices can be obtained by relational calculus [12].

The third and most popular approach is the interpretation as a graph reachability problem with the two steps of graph construction and performing a graph reachability analysis algorithm as the solution [13].

The representation of object-oriented programs turned out to become an important problem. New forms of dependency graphs have been proposed in order to represent concepts such as polymorphism, dynamic binding and inheritance namely in [14] and [15].

With the introduction of aspect-oriented programming existing graph representations for program slicing had to be extended as well. A dependence-based representation called aspect-oriented system dependence graph has been proposed in [16], consisting of three parts: a group of dependence graphs for aspect code, a system dependence graph for non-aspect code and additional dependence arcs used to connect the two aforementioned data structures. [17] developed a program slicing system using the AspectJ framework for Java and also described the benefits, usability and cost effectiveness of dynamic analysis through AOP and program slicing.

Considering the problem of slicing concurrent object-oriented programs another extension of previous program dependence representation has been proposed in [18], the so called system dependence net. It consists of a collection of procedure dependent nets each representing a procedure or method in the program and some additional arcs for the relations between the call, the called method and other data dependencies.

Regarding the big picture of software engineering, there has been research on the impact of program slicing in traditional software development processes. [6] especially addresses its unique importance in reducing cost and time when incorporated into modern software engineering phases such as testing, debugging and maintenance based on empirical studies.

In Debugging, dynamic slicing is preferred since the programmer is only interested in the faulty execution, instead of all executions [19]. Agrawals discussion on how static and dynamic slicing can be used for semi-automatic debugging [20] also proposed an approach where the user gradually zooms out from the point in code where the bug manifested by repeatedly considering larger data and control slices.

Duesterwald et al. [21] introduced program slicing into software testing with the result of a data-flow testing criterion that combines reasonable efficiency with reasonable precision. The research on applying program slicing to the problem of reducing the cost of regression testing can be divided into three groups. One group focusing on dynamic slicing, the second focusing on program dependence graphs and the third one basing its research around Weisers original data flow definition of slicing.

There have been various attempts at variation of the traditional slicing idea such as chopping [22] and dicing [23]. Chopping identifies dependency chains from a source to the sink of a dependence and tries to find the program part that causes one specific variable s to affect another specific variable t. Dicing on the other hand tries to find the difference between static program slices for two variables. It is often used for debugging when computation on some variables fails while computation on other variables succeeds.

3 Overview

We compute slices using a slicing criterion, a pair of a program location and a set of variables. In the remainder of this section we will give an example driven overview of program slicing (examples inspired by [7]).

original program | **program slice** criterion: $(2, \{y\})$

```
1 x = 5;                  1 x = 5
2 y = 10;                 2 y = 10;
```

As we observe, programs are always a legal program slice of itself. Usually one is interested in the smallest program slices which behaves semantically equivalent to the original program in respect to the variables, given in the slicing criterion. In the previous example the slice:

```
1
2 y = 10;
```

would be such a minimal slice.
In presence of control flow structures, their semantics have to be taken care of when computing program slices:

original program | **program slice** criterion: $(8, \{c\})$

```
1 a = 2;                  1 a = 2;
2 while (P)               2
3 {                       3 {
4   a = b - 2;            4  a = b - 2;
5   x = 10;               5
6   b = 64;               6  b = 64;
7 }                       7 }
8 c = a + 2;              8 c = a + 2
```

In this example P is an arbitrary predicate expression. If P holds at least one time (4) depends on (6). In other words the loop carries data dependencies into the next iteration. If P never holds, (8) has a data dependency to (1) - so (1) also contributes to the slice. So when computing slices *data dependencies* must be taken into account.

original program

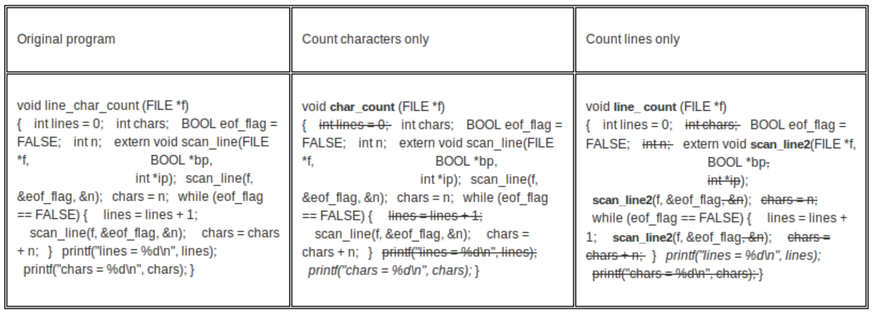
```
1  a = 10;
2  if(P)
3  {
4    z = 5;
5    if (Q)
6    {
7     a = 12;
8    }
9  }
10 c = c + a;
```

program slice
criterion: (10, $\{c\}$)

```
1  a = 10;
2  if(P)
3  {
4
5    if (Q)
6    {
7     a = 12;
8    }
9  }
10 c = c + a;
```

Also control flow elements transitively contribute to program slices. (10) depends on (7) and therefore also on (5) and transitively (2) (note that (4) is not in the slice).

Equipped with this knowledge we will now try to decompose the UNIX word-count tool into programs counting characters and lines separately:



Original program	Count characters only	Count lines only
void line_char_count (FILE *f) { int lines = 0; int chars; BOOL eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOL *bp, int *ip); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { lines = lines + 1; scan_line(f, &eof_flag, &n); chars = chars + n; } printf("lines = %d\n", lines); printf("chars = %d\n", chars); }	void **char_count** (FILE *f) { ~~int lines = 0;~~ int chars; BOOL eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOL *bp, int *ip); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { ~~lines = lines + 1;~~ scan_line(f, &eof_flag, &n); chars = chars + n; } ~~printf("lines = %d\n", lines);~~ *printf("chars = %d\n", chars);* }	void **line_ count** (FILE *f) { int lines = 0; ~~int chars;~~ BOOL eof_flag = FALSE; ~~int n;~~ extern void **scan_line2**(FILE *f, BOOL *bp~~,~~ ~~int *ip~~); **scan_line2**(f, &eof_flag~~, &n~~); ~~chars = n;~~ while (eof_flag == FALSE) { lines = lines + 1; **scan_line2**(f, &eof_flag~~, &n~~); ~~chars = chars + n;~~ } *printf("lines = %d\n", lines);* ~~printf("chars = %d\n", chars);~~ }

Table 1. Program slices of UNIX word-count utility[1]

[1] example taken from: http://www.grammatech.com/research/papers/slicing/slicingWhitepaper.html , accessed on 5.May 2012

In each slice, the starting point is shown in italics, and the striker regions indicate which program elements are to be removed from the original program.[2] Note that new names for external symbols have to be introduced (and also implemented.

Up to now computing slices answered the question which program parts effect the 'selected one'. Precicely spoken these slices are called *Backward-slices*. *Forward-slices* in contrary desribe what computations are effected by the selected one. Forward slices in general represent no executable program thus they are not applicable for decomposition. However forward slicies pose another interesting feature when it comes to program modifications - i.e. detecting defencies or bugs introduced by modifications in initialization code becomes easier.

original program

```
sum = 0;
x = 0;
while(i < 10)
{
 sum = sum + i;
 i++;
}
```

forward-slice
criterion: $(0, \{sum\})$

```
sum = 0;



 sum = sum + i;
```

4 Background

In this section we will give a short overview on the foundations of program slicing. For in depth analysis we refer to numerous surveys published in this field [9].

Program slices are usually defined using a *slicing criterion*. According to Mark Weiser [10] a slicing criterion for a program specifies a window for observing its behaviour. This window virtually marks a program point of interest, at which values of relevant variables should be investigated. Formally the slicing criterion is given by a pair (p, V) whereby p is a program point and V a subset of the variables which are in scope at program point p. Given this a program slice is defined by all program parts which are affected by p or have an effect on p respective to V.

It shall be noted that different formulations of slicing criteria have been proposed - however for our purposes the previously given one suffices. It is known that in general it is impossible to find the minimal slice given a target variable at some program point [10]. However several approximation algorithms for finding near optimal program slices have been proposed. There is control flow analysis [10], information flow relations [12] and by using a dependency graph [13].

In the remainder of this section we will focus on slicing by control flow analysis 4.1 and the dependency graph based approach 4.1.

4.1 Implementation techniques

Implementation by Control Flow Analysis
As noted by Weiser computing program slices can be seen as control flow analysis similar to the 'reaching definitions' problem (as described in [24]). The proposed algorithm works as in two phases:

– At each statement s compute a set of all variables R which effect variables used at statement s. As with other *forward analyses* the control flow graph is traversed in a top down manner whereby at each statement its dependencies get computed. To address loops the computation is carried out as fixed point algorithm - that is the resulting values for each statement get computed as long as the values change [3]. As the next example shows, this is not quite sufficient for computing program slices.

[3] We considered printing the precise equations but considered that to be out of scope

PC	Statement	R
0	$X < -READ$	$\{\}$
1	$IF\ X\ DO$	$\{(0, X)\}$
2	$Y = X$	$\{(0, X)\}$
3	$ENDIF$	$\{(0, X)(2, Y)\}$
4	$PRINT(Y)$	$\{(0, X)(2, Y)\}$

Actually $(4, Y)$ has a dependency to $(1, X)$ also which is not tracked. Of course statement 1 should be included in the resulting slice, so the algorithm has to address this in a second phase.

- In a second step Weiser computes all indirect effects onto variables used in *s*.
- As a third step the resulting program slice is computed by aggregation of all statements which affect the variables of interest at statement *s*.

Implementation utilizing the Program Dependency Graph

A program dependency graph (PDG) [25] may be seen as extension to control flow graphs. Control flow graphs make the control flow explicit, however retrieving data dependencies needs further computation. PDGs in contrast make both explicit, the control flow and data dependencies. In other words the PDG builds on a control flow graph but splits up each original node into nodes relevant for data dependencies (that is variables and predicates). For illustration purpose we give a simple *C* Program computing sums. The resulting PDG is depicted in Figure 1.

```
void main() {
    int i = 1;     int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

```

```

static int add(int a, int b)
{
    return(a+b);
}
```

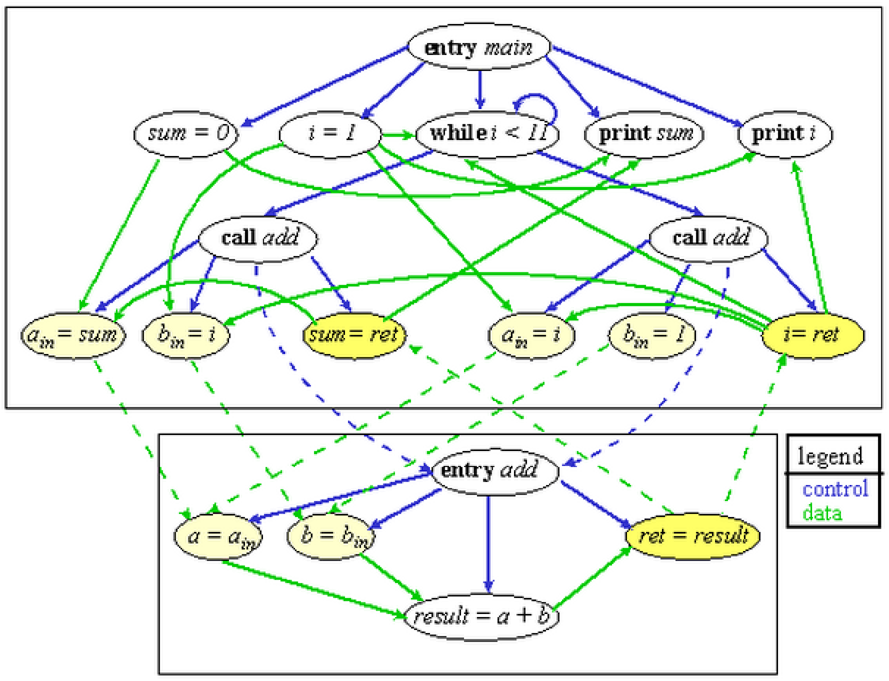


Fig. 1. Program slices of UNIX word-count utility[4]

Given a PDG retrieving program slices reduces in traversing the graph backwards, following dependencies. Each traversed node is therefore added into the slice continuously. Aside from simplicity the algorithms capability to work interpretively shows another advantage of this method.

[4] example taken from: http://www.grammatech.com/research/papers/slicing/slicingWhitepaper.html , accessed on 5.May 2012

5 Program Slicing in the Software Lifecycle

The following pages describe some applications of program slicing in the software lifecycle. The goal of using program slicing in these applications is, to support developers and testers in their daily work and to give them a better overview of the program they are working on.

5.1 Debugging and Program Analysis

One of the main scope of applications for software slicing is the field of debugging software and program analysis. The main problem at debugging software is, that most software covers thousand lines of code and can be very confusing, especially if the program is written long time ago. By the use of program slicing, the amount of lines, that the developer has to inspect, can be decreased dramatically. In order to see only the important parts of the program for the current debugging tasks, developers can slice away the uninteresting parts of the software and are so able to concentrate on the buggy functions or lines of code.

Kusumoto et al. [26] show a simple experiment in order to evaluate the fault localization supported by program slicing. The results show, that the number of lines, that have to be checked, can be reduced by more than half of the initial amount. Further on they show, that the average time for finding a fault in a given program can be reduced by an average of 30%. The full list of results is shown in table 5.1 on page 11.

Table 2. Fault localization results according to Kusumoto [26]

	try	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**total**
Without Slicing	1	17	10	26	27	35	17	7	139
	2	28	18	36	17	25	23	16	163
	3	23	12	28	32	41	17	7	160
With Slicing	1	38	6	27	18	20	20	8	119
	2	11	8	10	16	16	13	7	88
	3	14	14	19	36	10	5	17	116

5.2 Software Maintenance

Up to now we have seen that program slicing drastically suppports debugging and program analysis. However, since Software maintenance is the longest and possibly most expensive part in the software cycle we now investigate program slicing in respect to software maintenance more closely.

Although applications in software maintenance strongly build on program analysis we consider program slicing in software maintenance as notable distinctive area. Saleem et al. [6] differentiate between software re-engineering and software comprehension. According to their work program slicing in the field of software maintenance is limited to identifying modified parts of the software and to assure a perfect understanding of the software by dividing it into program slices. They also provide some numbers for 43 different c-programs, which show, that the reduction of lines of code by the application of program slicing lies between 7.4% and 48.8%. The effect of comprehension assisted by slicing on overall software budget and time lies between 3.6% and 26.1%. Table 3 on page 13 shows (a representative excerpt) the results of the experiment.

Binkley and Gallagher [7] describe the application of program slicing in software maintenance as follows. The whole software is divided into independend, dependent and compliment parts by program slicing. After dividing the program into slices, all variables can be categorized as changeable, unchangeable and used. If a developer changes parts of the code, only the code in the independent and dependent parts is important. If the developer only changes variables that are categorized as changeable, the complement part of the program is definitely unaffected by the modifications, what means, that it is not necessary to test the complement part of the program again after these modifications.

5.3 Software Testing

During the software development but also during its maintenance, software testing is one of the most important tasks in order to guar-

Table 3. The effects of program slicing in software comprehension as provided by [6]

Program	**LOC before prog. slic.**	**LOC after prog. slic.**	**Percent reduction LOC**	**Affect on overall software budget, time**
acct-6.3	6,764	501	30.5%	3.6%
bc	11,173	5,452	48.8%	23.9%
byacc	5,501	996	18.1%	8.9%
cadp	10,620	828	7.8%	3.8%
copia	1,112	252	22.7%	11.1%
csurf-pkgs	38,507	6,007	15.6%	7.6%
cvs	67,828	31,404	46.3%	22.7%
diffutils	12,705	2,871	22.6%	11.1%
ed	9,046	4,822	53.5%	26.1%
empire	48,800	16,104	33.0%	16.2%
espresso	21,780	6,708	30.8%	15.1%
findutils	11,843	3,257	27.5%	13.5%
flex2-5-4	15,283	3,194	20.9%	10.2%
ftpd	15,361	5,361	34.9%	17.1%
gcc.cpp	5,731	2,625	45.8%	22.4%

antee the functionality and the quality of program. Although most research papers only describe the application of software testing in generally, they all are about regression testing.

Regression testing is the field of software testing where the program is tested after adding some changes to it. The goal of regression testing is, that no new bugs get introduced due to subsequent program modifications. Since running tests on a complete program is a very costly and time-consuming process, it is desirable to know, which parts of the program are unaffected by modifications and which test-cases can be omitted, without compromising the quality of the software.

By the application of program slicing, all statements in a program can be categorized as unaffected or affected statements according to the modified part of the software. After modificating the source code, only these parts of the program have to be tested again, that are part of the affected statements. According to Saleem et al. [6] this approach leads to a reduced test case computation time of 71% and saves about 85% of the costs for fixing a bug internally.

Another approach for program slicing supported regression testing is the automated selection of test-cases, that have to executed after a modification. Gallagher and Binkley [7] describe algorithms, that are able to select the important test cases after applying program slicing to the software and the part, that is going to be modified.

5.4 Program Integration

Although it can be seen as a part of software maintenance, Gallagher and Binkley [7] describes the application of program slicing in the field of program integration explicitly.

The problem about program integration is, that two or more variants of a program have to be merged, so that the resulting program combines the functionality of all variants in the correct way. In a big software system it is hard to check, if the functionality and quality of the combined program is the same after the program integration, as it was before. These problems often appear during distributed development and therefore relevant to the modern software development approach.

By using program slicing, the differing parts between the base program and two or more variants can be extracted. The resulting program slices can be checked for differences. Components of the base application which aren't changed in any of the variants can be seen as preserved. The other parts of the program are merged by analysing the differences in the relevant slices. If the resulting program hasn't got interferences, which are incompatible changes in two or more variants, it combines the functionality of the base program and all of the merged variants.

5.5 Software Quality Assurance

Gallagher and Binkley [7] describe an application of program slicing in the field of software quality assurance. In order to identify safety critical components in a software system, program slicing can be used to create slices, that contain all statements, which affect these critical components. In this manner a perspective of the source code can be generated, which allows the developer a better view onto the important components and functions of the software.

The approach supports the finding of common mode failures, which are failures that arise from a common cause. For example the call of a simple procedure in two different safety critical functions may cause an error in the system, if the simple procedure is implemented incorrectly. By using program slicing and backward slices, the simple procedure will appear in the resulting slices and the developer therefor knows, that the simple procedure is important too.

5.6 Reverse Engineering

Another application for program slicing is the field of reverse engineering. In order to analyse the functionality of an old program it might be necessary to reverse engineer parts of it. By applying program slicing to the source code the developer is supported in his work, as it gets easier to see the important parts of the program. For example, the analysis of different versions of a program, gets easier, as program slicing supports finding the differences between the two programs.

According to [7], a special kind of program slicing called interface slicing can be used for reverse engineering. Interface slicing allows the programmer to define, which exported functions are of relevance and the resulting slices represents the public interfaces of the program without revealing the source for their implementation.

6 Practical Implementations and Tools

We have shown that program slicing is beneficial to all stages of software development. The authors were quite surprised of the quite thin landscape in production ready slicing tools. However in this section we will briefly review some current implementations and tools.

As argued by [27] practical implementations of languages as C and C++ pose several challenges - as dangling references and aliasing, exceptions and external libraries. However some tools for C++, ANSI C and Oberon have been implemented. [5]

Ranganath et. al [27] propose a eclipse plugin for the java programming language called Kaveri [6]. Its implementation is based on Indus, a static analysis and transformation tool for transforming java programs.

A dynamic slicing tool for the java programming language is JSlice [7]. In contrast to Kaveri, JSlice works directly on JVM-Bytecode [29].

7 Conclusion

We have provided a gentle introduction into the field of program slicing. Furthermore we gave a short overview of implementation techniques and its theoretical background. When reviewing program slicing and its potential in software maintenance we discovered a much broader application area: Program slicing supports all phases in the software development cycle. It turned out to have a huge impact on Program comprehension and debbuging, software testing and of course software maintenance. Experiments have shown that Program size in some cases can be reduced by even 50% resulting in 30% less time needed to identify faulty code. Last but not least we reviewed some of current production quality implementations of program slicing for modern programming languages.

[5] we refer to Krinkes phd thesis for an overview of these implementations [28]

[6] http://marketplace.eclipse.org/content/kaveri-java-program-slicing-plugin

[7] http://jslice.sourceforge.net/

References

1. Weiser, M.D.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, Ann Arbor, MI, USA (1979) AAI8007856.
2. Weiser, M., Lyle, J.: Experiments on slicing-based debugging aids. In: Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers, Norwood, NJ, USA, Ablex Publishing Corp. (1986) 187–197
3. Agrawal, H., Demillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. Softw. Pract. Exper. **23**(6) (June 1993) 589–616
4. Korel, B., Rilling, J.: Program slicing in understanding of large programs. In: Proceedings of the 6th International Workshop on Program Comprehension. IWPC '98, Washington, DC, USA, IEEE Computer Society (1998) 145–
5. Korel, B., Rilling, J.: Dynamic program slicing in understanding of program execution. In: Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on. (mar 1997) 80 –89
6. Saleem, M., Hussain, R., Ismail, V., Mohsin, S.: Cost effective software engineering using program slicing techniques. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human. ICIS '09, New York, NY, USA, ACM (2009) 768–772
7. Gallagher, K., Binkley, D.: Program slicing (1996)
8. Lucia, A.D.: Program slicing: Methods and applications (2001)
9. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2) (March 2005) 1–36
10. Weiser, M.: Program slicing. In: Proceedings of the 5th international conference on Software engineering. ICSE '81, Piscataway, NJ, USA, IEEE Press (1981) 439–449
11. Jackson, D., Rollins, E.J.: A new model of program dependences for reverse engineering. In: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering. SIGSOFT '94, New York, NY, USA, ACM (1994) 2–10
12. Bergeretti, J.F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst. **7**(1) (January 1985) 37–61
13. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. SDE 1, New York, NY, USA, ACM (1984) 177–184
14. Larsen, L., Harrold, M.J.: Slicing object-oriented software. In: Proceedings of the 18th international conference on Software engineering. ICSE '96, Washington, DC, USA, IEEE Computer Society (1996) 495–505
15. Chen, J.L., Wang, F.J., Chen, Y.L.: Slicing object-oriented programs. In: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference. APSEC '97, Washington, DC, USA, IEEE Computer Society (1997) 395–
16. Zhao, J.: Slicing aspect-oriented software. In: Proceedings of the 10th International Workshop on Program Comprehension. IWPC '02, Washington, DC, USA, IEEE Computer Society (2002) 251–
17. Ishio, T., Kusumoto, S., Inoue, K.: Application of aspect-oriented programming to calculation of program slice (2003)
18. Zhao, J., Cheng, J., Ushijima, K.: Static slicing of concurrent object-oriented programs. In: Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference, IEEE Computer Society Press (1996) 312–320

19. Korel, B., Rilling, J.: Application of dynamic slicing in program debugging. In: In Proceedings of the Third International Workshop on Automatic Debugging (AADE-BUG '97), Linkoping. (1997)
20. Agrawal, H.: Towards automatic debugging of computer programs (1991)
21. Duesterwald, E., Gupta, R., Soffa, M.L.: Rigorous data flow testing through output influences. (March 1992) 131–145
22. Jackson, D., Rollins, E.J.: Chopping: A generalization of slicing. Technical report, In Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (1994)
23. Chen, T.Y., Cheung, Y.Y.: Dynamic program dicing. In: Proceedings of the Conference on Software Maintenance. ICSM '93, Washington, DC, USA, IEEE Computer Society (1993) 378–385
24. Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier Science Inc., New York, NY, USA (1977)
25. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (July 1987) 319–349
26. Kusumoto, S., Nishimatsu, A., Nishie, K., Inoue, K.: Experimental evaluation of program slicing for fault localization. Empirical Softw. Engg. **7**(1) (March 2002) 49–76
27. Ranganath, V.P., Hatcliff, J.: Slicing concurrent java programs using indus and kaveri. Int. J. Softw. Tools Technol. Transf. **9**(5) (October 2007) 489–504
28. Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. PhD thesis, Universität Passau (April 2003)
29. Wang, T., Roychoudhury, A.: Using compressed bytecode traces for slicing Java programs. In: ACM/IEEE International Conference on Software Engineering (ICSE). (2004) 512–521