Program Slicing: Methods and Applications

Andrea De Lucia

Faculty of Engineering
University of Sannio
Palazzo Bosco Lucarelli, Piazza Roma
82100 Benevento, Italy
delucia@unisannio.it

Abstract

Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program. Examples of applications include debugging, testing, program comprehension, restructuring, downsizing, and parallelization. This paper discusses different statement deletion based slicing methods, together with algorithms and applications to software engineering.

1 Introduction

Several software engineering tasks require reducing the size of programs or decomposing a larger program into smaller components. Examples include debugging, testing, program comprehension, restructuring, downsizing, and parallelization. Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program.

Program slicing was first introduced by Mark Weiser [84]. His research was motivated by the need to help students understand and debug their programs. Weiser discovered that the mental process made by programmers when debugging their code was slicing [85] and tried to formally define this process and its output [86]. Since then, program slicing has grown as a field and an amazing number of papers have been published that present different forms of program slicing, algorithms to compute them, and applications to software engineering. Different program slicing surveys have also been published [8, 49, 81], as well as journal special issues [34].

According to the original definition [86], the notion of slice was based on the deletion of statements. A slice is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point. Several variants of this notion have been proposed in the literature, such as dynamic slicing [58], quasi static slicing [83], simultaneous dynamic slicing [37], and conditioned slicing [14]. Forms of slicing that are based on amore general framework of transformations, including the simple statement deletion, have also been proposed [39]. This paper discusses statement deletion based slicing methods together with algorithms and applications to software engineering. The paper concludes with further issues on program slicing and directions for future work.

2 Static slicing

From a formal point of view the definition of a slice is based on the concept of slicing criterion. According to Weiser [86], a slicing criterion is a pair $<p, V>$, where p is a program point and V is a subset of program variables. A program slice on the slicing criterion $<p, V>$ is a subset of program statements that preserves the behavior of the original program at the program point p with respect to the program variables in V, i.e., the values of the variables in V at program point p are the same in both the original program and the slice. As the behavior of the original program has to be preserved on any input, Weiser's slicing has been called *static slicing*, to differentiate it from other forms of slicing that require the behavior be preserved on a subset of input to the program.

Weiser has demonstrated that computing the minimal subset of statements that satisfies this requirement is undecidable [84, 86]. However, an approximation can be found by computing the least solution to a set of data flow equations relating a Control Flow Graph (CFG) node to the variables which are relevant at that node with respect to the slicing criterion [86].

The algorithm proposed by Weiser originated an alternative and commonly adopted definition: a slice consists of the set of program statements and predicates that directly or indirectly affect the computation of the variables in V before the execution of p. According to this definition, a different algorithm has been proposed that computes slices as backward traversals of the Program Dependence Graph (PDG) [76], a program representation where nodes represents statements and predicates, while edges carry information about control and data

dependences. The PDG based algorithm considers slicing criteria of type <*p*, *V*>, where p is a program point and *V* is the set of variables referenced at *p*. A slice with respect to such a slicing criterion consists of the set of nodes that directly or indirectly affect the computation of the variables in *V* at node *p*[1]. This form of slice has also been defined *backward slice*, in contrast to a *forward slice*, defined as the set of program statements and predicates affected by the computation of the value of a variable *v* at a program point *p* [45].

Horwitz *et al.* [45] extended the PDG based algorithm to compute interprocedural slices on the System Dependence Graph (SDG). The authors demonstrated that their algorithm is more accurate than the original interprocedural slicing algorithm by Weiser [86], because it accounts for procedure calling contexts. Recent improvements of algorithms to compute slices through graph reachability are presented in [77].

A different parallel algorithm to compute slices has been presented by Danicic *et al.* [25]. The control flow graph of a program is converted into a network of concurrent processes whose parallel execution produces the slice. Algorithms have also been proposed that compute backward static slices in the presence of arbitrary control flow [2, 5, 20, 40, 80] and pointers [69, 68, 67, 72].

Different applications of static slicing have been proposed in the literature, with some variants on the original definition. For example, Gallagher and Lyle [33] introduced the concept of *decomposition slicing* and discussed its application to software maintenance. A decomposition slice is defined with respect to a variable *v*, independently of any program point. It is given by the union of the static slices computed with respect to the variable *v* and all possible program points *p*.

Other applications of program slicing include software testing [9, 36, 42, 38, 43], program debugging [85, 70], measurement [75, 7, 73, 74], validation [63], program parallelization [86], program integration [44], reverse engineering and program comprehension [6, 29], program restructuring [52, 16, 21, 64, 51], and identification of reusable functions [17, 22, 65]. In particular, the identification of an internal reusable function requires a different definition of slicing criterion that allows the algorithm to stop the computation of the slice whenever the searched function has been identified. Weiser's slicing criterion only provides the end point and the set of output variables of the function to be extracted. Lanubile and Visaggio [65] added the set of input variables of the searched function to the slicing criterion. They introduced the notion of *transform slice*, as the slice that computes the values of the output variables at a given program point from the values of the input variables. The computation of a transform slice is similar to the computation of a static backward slice but stops as soon as the statements that define values for the input variables are included in the slice. On the other hand, Cimitile *et al.* [21, 22] defined a different slicing criterion including both the start and the end statements of the function to be extracted. The slice is computed between these two statements that form a one-in/one-out subgraph of the CFG. The authors also defined a method to identify the slicing criterion from the specification of the searched function expressed in terms of a precondition and a postcondition.

[1] Recent implementations of the algorithm enables slicing on single used variables at node *p* (D. Binkley, private communication, 2001).

3 Dynamic slicing

Program slicing has first been proposed as a tool for decomposing programs during debugging, in order to allow a better understanding of the portion of code which revealed an error [85, 86]. In this case the slicing criterion contains the variables which produced an unexpected result on some input to the program. However, a static slice may very often contain statements which have no influence on the values of the variables of interest for the particular execution in which the anomalous behavior of the program was discovered.

Korel and Lasky [57, 58] proposed an alternative slicing definition, namely *dynamic slicing*, which uses dynamic analysis to identify all and only the statements that affect the variables of interest on the particular anomalous execution trace. In this way, the size of the slice can be considerably reduced, thus allowing an easier localization of the bugs. Another advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire array [46]. Similarly, dynamic slicing distinguishes which objects are pointed to by pointer variables during a program execution.

Korel and Lasky [58] proposed an iterative algorithm based on data flow equations to compute dynamic slices. The algorithm requires that if any occurrence of a statement (within a loop) in the execution trace is included in the slice, then all the other occurrences of that statement be included in the slice. This ensures that the slice extracted is executable.

Other algorithms proposed in the literature produce slices that are not executable, because they are not necessarily executable subsets of the original program [1, 35]. In particular the algorithm by Agrawal and Horgan [1] uses dynamic dependence graphs to produce more refined slices. It considers only the occurrences of statements in the trajectory that affect the computation of the variables in the slicing criterion. Interprocedural slicing algorithms based on dependence graphs have also been proposed [47] as well as dynamic slicing algorithms in the presence of unconstrained pointers [3] and arbitrary control flow [55].

Besides debugging [58, 50, 4], dynamic slicing has been used for several applications, including software testing [48], software maintenance [54, 60], and program comprehension [59]. A survey and comparison of dynamic slicing methods has been presented by Korel and Rilling [61].

4 Quasi static slicing

Quasi static slicing was the first attempt to define a hybrid slicing method ranging between static and dynamic slicing [83]. The need for quasi static slicing arises from applications where the value of some input variables is fixed while the behavior of the program must be analyzed when other input values vary. Indeed, a quasi static slice preserves the behavior of the original program with respect to the variables of the slicing criterion, on a subset of the possible program inputs. This subset is specified by the possible combination of values that the unconstrained input variables might assume. Of course, in the case all variables are unconstrained, the quasi static slice coincides with a static slice, while when the values of all input variables are fixed, the slice is a dynamic slice.
The notion of quasi static slicing is closely related to *partial evaluation* or *mixed computation* [11], a technique to specialize programs with respect to partial inputs. By specifying the values of some of the input variables, constant propagation and simplification can be used to reduce expressions to constants. In this way, the values of some program predicates can be evaluated, thus allowing the deletion of branches which are not executed on the particular partial input. Quasi static slices are computed on specialized programs.
Quasi static slicing has been applied for program comprehension together with transformations [41]. From this point of view, this approach can be considered as an extension of work presented in [12] where partial evaluation is used to understand programs. Combining partial evaluation with program slicing allows to restrict the focus of the specialized program with respect to a subset of program variables and a program point.

5 Simultaneous dynamic slicing

A different form of slicing introduced by Hall [37] computes slices with respect to a set of program executions. This slicing method is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases, rather than just one test case. A simultaneous program slice on a set of test cases is not simply given by the union of the dynamic slices on the component test cases. Indeed, simply unioning dynamic slices is unsound, in that the union does not maintain simultaneous correctness on all the inputs. Hall [37] proposed an iterative algorithm that, starting from an initial set of statements, incrementally builds the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.
Simultaneous dynamic slicing has been used to locate functionality in code. The set of test cases can be seen as a kind of specification of the functionality to be identified. This approach can also be seen as an extension of the approach by Wilde *et al.* [87, 88], where test cases are used to identify the set of source code statements implementing a functionality. Combining slicing with this approach results in a more precise identification of the functionality to be extracted.

6 Conditioned slicing

Conditioned slicing is a general framework for statement deletion based slicing [14]. A conditioned slice consists of a subset of program statements which preserves the behavior of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterize these executions is specified in terms of a first order logic formula on the input.
Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyze code fragments with respect to different perspectives. Canfora *et al.* [14] have demonstrated that conditioned slicing subsumes any other form of statement deletion based slicing method, i.e., the conditioned slicing criterion can be specified to obtain any form of slice.
A conditioned slice can be computed by first simplifying the program with respect to the condition on the input (i.e., discarding *infeasible paths* with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [53, 24] can be used to compute the reduced program, also called *conditioned program* in [15]. Although the identification of the infeasible paths of a conditioned program is in general an *undecidable* problem, in most cases implications between conditions can be automatically evaluated by a theorem prover, e.g. [13]. In [14] conditioned slices are interactively computed: the software engineer is required to make decisions the symbolic executor cannot make. An automatic conditioned slicer has been implemented by Danicic *et al.* [26].
Different variants of conditioned slicing have been presented in the literature [71, 31]. Ning *et al.* [71] proposed a tool, called COBOL/SRE, to extract different types of slices from legacy systems, in particular *condition-based slices*. The user specifies a logical expression and a slicing range and the tool automatically isolate the statements that can be reached along control flow paths under the given condition. However, the authors did not propose a formal definition of condition-

based slicing. Field *et al.* [31] introduced the concept of *constrained slice* to indicate slices that can be computed with respect to any set of constraints. Their approach is based on an intermediate representation for imperative programs, named PIM, and exploits graph rewriting techniques based on *dynamic dependence tracking* [32] that model symbolic execution. The slices extracted are not executable. The authors are interested in the semantic aspects of more complex program transformations rather than in simple statement deletion.
An extension of conditioned slicing, namely *backward conditioning*, has been proposed by Danicic *et al.* [27]. While conditioned slicing uses forward conditioning and deletes statements that are not executed when the initial state satisfies the condition, backward conditioning deletes statements which cannot cause execution to enter a state which satisfies the condition. Backward conditioning addresses questions of the form "what parts of the program could potentially lead to the program arriving in state satisfying a given condition ?", whereas forward conditioning addresses questions of the form "what happens if the program starts in a state satisfying a given condition ?" [27].
Conditioned slicing has been applied to program comprehension [28, 27] and to the extraction of reusable functions [15, 71]. The use of symbolic execution to specialize generalized software components to more specific and efficient functions to be used under more restricted conditions has been proposed by Coen-Porisini *et al.* [23]. In this sense conditioned slicing can be considered an extension of this approach.

7 Other slicing methods and future directions

This paper has surveyed statement deletion based methods for programs written in procedural languages. A number of slicing resources are available on the web (a good entry point is Jens Krinke's web page[2]), including large scale slicing research tools (see for example tools developed within the Wisconsin[3] and the Unravel[4] slicing projects).
Concerning the proposed applications of slicing, most of them are related to software testing and debugging and to software maintenance tasks, like for example program comprehension and restructuring. The introduction of new distributed technologies calls for applications of slicing to program parallelization or migration to distributed architectures. Mark Weiser was a precursor of this issue in his seminal paper [86] and in his foreword to the special issue on program slicing [34] again pointed out the need for producing a major research effort in this direction. At the present, few contributions have been proposed. An example is the method presented by Canfora *et al.* [16], where a control dependence based slicing algorithm is used to decompose legacy programs into client-server components.
The wide spread of new object-oriented and distributed technologies also calls for new slicing algorithms to be applied to the new programming languages. From this point of view, a research effort has already been produced and several contributions to the definition of slicing methods for object-oriented [66, 82, 18, 79], concurrent, and distributed [30, 19, 56, 62, 89] software applications have already been published, but research should also concentrate on slicing web-based applications developed with heterogeneous programming languages and technologies. First attempts in this direction are recently being made [78].
A final point of concern is what should the term slicing mean. The framework of statement deletion based slicing methods [14] can be extended with more powerful simplification rules. Harman and Danicic [39] introduced *amorphous program slicing*. Like a traditional slice, an amorphous slice preserves a projection of the semantics of the original program from which it is constructed. However, it can be computed by applying a broader range of transformation rules, including statement deletion. This is particularly useful in program comprehension, as more powerful transformations may sensibly simplify complex programs [10]. Other applications of this approach include the extraction of reusable functions and program parallelization. Indeed, it is worth noting that the slices extracted by removing not relevant statements might contain code fragments used for computing intermediate results and common to the different slices extracted from the original program [75, 33]. Transformation rules applied to these slices can make the extracted program components more self-contained and more understandable in future maintenance.

[2] http://www.infosun.fmi.uni-passau.de/st/staff/krinke/slicing/
[3] http://www.cs.wisc.edu/wpis.html
[4] http://hissa.ncsl.nist.gov/~jimmy/unravel.html

Acknowledgments

I would like to thank Mark Harman for his comments on a previous version of this paper.

References

[1] H. Agrawal and J.R. Horgan, "Dynamic program slicing", *Proceedings of ACM SIGPLAN Conference on programming Language Design and Implementation*, White Plains, New York, U.S.A., ACM Press, 1990, pp. 246-256.
[2] H. Agrawal, "On slicing programs with jump statements", *ACM Sigplan Notices*, vol. 29,no. 6, 1994, pp.302-312.

[3] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers", *Proceedings of 4th ACM Symposium on Testing, Analysis, and Verification*, 1991, pp. 60-73.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking", *Software – Practice and Experience*, vol. 23, no. 6, 1993, pp. 589-616.

[5] T. Ball and S. Horwitz, "Slicing programs with arbitrary control-flow", *Proceedings of 1st International Workshop on Automated and Algorithmic Debugging*, Linköping, Sweden, Springer Verlag, New York, 1993, pp. 106-222.

[6] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering", *Proceedings of 15th International Conference on Software Engineering*, Baltimore, Maryland, USA, IEEE CS Press, 1993, pp. 509-518

[7] J.M. Bieman and L.M. Ott, "Measuring functional cohesion", *IEEE Transactions on Software Engineering*, vol. 20, no. 8, 1994, pp. 644-657.

[8] D. Binkley and K.B. Gallagher, "Program slicing", in *Advances in Computers*, vol. 43, Marvin Zelkowitz, Editor, Academic Press, San Diego, CA, 1996, pp. 1-52.

[9] D. Binkley, "The application of program slicing to regression testing", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 583-594.

[10] D Binkley, M Harman, L.R. Raszewski, and C. Smith, "An empirical study of amorphous slicing as a program comprehension support tool", *Proceedings of 8th International Workshop on Program Comprehension*, Limerick, Ireland, 2000, IEEE CS Press, pp. 161-170.

[11] D. Bjorner, A.P. Ershov, and N.D. Jones, *Partial evaluation and mixed computation*, North-Holland, 1987.

[12] S. Blazy and P. Facon, "Partial evaluation for the understanding of FORTRAN programs", *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, 1994, pp. 535-559.

[13] R.S. Boyer and J.S. Moore, *A Computational Logic*, Academic Press, New York, 1979.

[14] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned program slicing", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 595-607.

[15] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, "Software salvaging based on conditions", *Proceedings of International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE CS Press, 1994, pp. 424-433.

[16] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, "Decomposing legacy programs: a First step towards migrating to client-server platforms", *The Journal of Systems and Software*, vol. 54, 2000, pp. 99-110.

[17] G. Canfora, A. De Lucia, G.A. Di Lucca, and A. R. Fasolino, "Slicing large programs to isolate reusable functions", *Proceedings of EUROMICRO Conference*, Liverpool, U.K., 1994, IEEE CS Press, pp. 140-147.

[18] J.L. Chen, F.J. Wang, and Y.L. Chen, "Slicing object-oriented programs", *Proceedings of APSEC'97*, Hong Kong, China, 1997, pp. 395-404.

[19] J. Cheng, "Slicing concurrent programs –a graph theoretical approach", *Proceedings of 1st International Workshop on Automated and Algorithmic Debugging*, Linköping, Sweden, Springer Verlag, New York, 1993, pp. 232-245.

[20] J.D. Choi and J. Ferrante, "Static slicing in the presence of goto statements", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, 1994, pp. 1097-1113.

[21] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study", *Proceedings of International Conference on Software Maintenance*, Opio (Nice), France, IEEE CS Press, 1995, pp. 124-133.

[22] A. Cimitile, A. De Lucia, M. Munro, "A specification driven slicing process for identifying reusable functions", *Journal of Software Maintenance: Research and Practice*, vol. 8, n. 3, 1996, pp. 145-178.

[23] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli, "Software specialization via symbolic execution", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, 1991, pp. 884-899.

[24] P.D. Coward, "Symbolic execution systems - a review", *Software Engineering Journal*, vol. 3, no. 6, 1988, pp. 229-239.

[25] S. Danicic, M. Harman, and Y. Sivagurunathan, "A parallel algorithm for static program slicing", *Information Processing Letters*, vol. 56, no. 6, 1995, pp. 307-313.

[26] S. Danicic, C. Fox, M. Harman, R. Hierons, "ConSIT: a conditioned program slicer", *Proceedings of International Conference on Software Maintenance*, S.José, USA, 2000, IEEE CS Press, pp. 216-226.

[27] S. Danicic, C. Fox, M. Harman and R. Hierons, "Backward Conditioning: a new program specialisation technique and its application to program comprehension", *Proceedings of 9th International Workshop on Program Comprehension*, Toronto, Canada, IEEE CS Press, 2001, pp. 89-97.

[28] A. De Lucia, A.R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing", *Proceedings of 4th Workshop on Program*

Comprehension, Berlin, Germany, IEEE CS Press, 1996, pp. 9-18.

[29] Y. Deng, S. Kothari, and Y. Namara, "Program slicing browser", *Proceedings of 9th International Workshop on Program Comprehension*, Toronto, Ontario, Canada, IEEE CS Press, 2001, pp. 50-59.

[30] E. Duesterwald, R. Gupta, and M.L. Soffa, "Distributed slicing and partial re-execution for distributed programs", *Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, New Haves, Connecticut, USA, 1992, pp. 329-337.

[31] J. Field, G. Ramalingan, and F. Tip, "Parametric program slicing", *Conference Record of the 22nd ACM Symposioum on Principles of Programming Languages*, San Francisco, California, U.S.A., 1995, pp. 379-392.

[32] J. Field and F. Tip, "Dynamic dependence in term of rewriting systems and its application to program slicing", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 609-636.

[33] K.B. Gallagher and J.R. Lyle, "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991, pp. 751-761.

[34] K.B. Gallagher and M. Harman (ed.), "Program slicing", special issue, *Information and Software Technology*, vol. 40, no. 11/12, 1998.

[35] R.Gopal, "Dynamic program slicing based on dependence relations", *Proceedings of Conference on Software Maintenance*, Sorrento, Italy, IEEE CS Press, 1991, pp. 191-200.

[36] R. Gupta, M.J. Harrold, and M.L. Soffa, "An approach to regression testing using slicing", *Proceedings of the Conference on Software Maintenance*, Orlando, FL, U.S.A., IEEE CS Press, 1992, pp. 299-308.

[37] R.J. Hall, "Automatic extraction of executable program subsets by simultaneous program slicing", *Journal of Automated Software Engineering*, vol. 2, no. 1, 1995, pp. 33-53.

[38] M. Harman and S. Danicic, "Using Program Slicing to Simplify Testing", *Journal of Software Testing, Verification and Reliability*, vol 5, no. 3, 1995, pp. 143-162.

[39] M. Harman and S. Danicic, "Amorphous program slicing", *Proceedings of 5th International Workshop on Program Comprehension*, Dearborn, Michigan, U.S.A., IEEE CS Press, 1997, pp.~70-79

[40] M. Harman and S. Danicic, "A new algorithm for slicing unstructured programs", *Journal of Software Maintenance: Research and Practice*, vol. 10, no. 6, 1998, pp. 415-441.

[41] M. Harman, S. Danicic, and Y. Sivagurunathan, "Program comprehension assisted by slicing and transformation", *Proceedings of 1st UK Program Comprehension Workshop*, Durham, UK, 1995.

[42] M.J. Harrold and M.L. Soffa, "Selecting data-flow integration testing", *IEEE Software*, vol 8, no. 2, 1991, pp. 58-65.

[43] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants", *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 4, 1999, pp. 233-262.

[44] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs", *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, 1989, pp. 345-387.

[45] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, 1990, pp. 26-60.

[46] J. Jiang, X. Zhou, and D.J. Robson, "Program slicing for C - the problems in implementation", *Proceedings of Conference on Software Maintenance*, Sorrento, Italy, IEEE CS Press, 1991, pp. 182-190.

[47] M. Kamkar, P. Fritzson, and N. Shahmerhi, "Three approaches to interprocedural dynamic slicing", *EUROMICRO Journal of Microprocessing and Microprogramming*, vol. 38, 1993, pp. 625-636.

[48] M. Kamkar, P. Fritzson, and N. Shahmerhi, "Interpocedural dynamic slicing applied to interprocedural data flow testing", *Proceedings of Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE CS Press, 1993, pp. 386-395.

[49] M. Kamkar, "An overview and comparative classification of program slicing techniques", *The Journal of Systems and Software*, vol. 31, 1995, pp. 197-214.

[50] M. Kamkar, "Application of program slicing in algorithm debugging", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 637-646.

[51] B.K. Kang and J.M. Bieman, "Using design abstractions to visualize, quantify, and restructure software", *Journal of Systems and Software*, vol. 42, no. 2, 1998, pp. 175-187.

[52] H.S. Kim and Y.R. Kwon, "Restructuring programs through program slicing", *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 3, 1994, pp. 349-368.

[53] J.C. King, "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, 1976, pp. 385-394.

[54] B. Korel, "Identifying faulty modifications in software maintenance", *Proceedings of 1st International Workshop on Automated and Algorithmic Debugging*, Linköping, Sweden, Springer Verlag, New York, 1993, pp. 341-356.

[55] B. Korel, "Computation of dynamic slices for unstructured programs", *IEEE Transactions on Software Engineering*, vol. 23, no. 1, 1997, pp. 17-34.

[56] B. Korel and R. Ferguson, "Dynamic slicing of distributed programs", *Journal of Applied Mathematics and Computer Science*, vol. 2, no. 2, 1992, pp. 199-215.

[57] B. Korel and J. Laski, "Dynamic program slicing", *Information Processing Letters*, vol. 29, no. 3, 1988, pp. 155-163.

[58] B. Korel and J. Laski, "Dynamic slicing of computer programs", *The Journal of Systems and Software*, vol. 13, no. 3, 1990, pp. 187-195.

[59] B. Korel and J. Rilling, "Dynamic program slicing in understanding of program execution", *Proceedings of 5th International Workshop on Program Comprehension*, Dearborn, MI, USA, 1997, pp. 80-90.

[60] B. Korel and J. Rilling, "CASE and dynamic program slicing in software maintenance", *International Journal of Computer Science and Information Management*, June 1998.

[61] B. Korel and J. Rilling, "Dynamic program slicing methods", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 647-659.

[62] J. Krinke, "Static slicing of threaded programs", *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, Montreal, Quebec, Canada, 1998, pp. 35-42.

[63] J. Krinke and G. Snelting, "Validation of measurement software as an application of slicing and constraint solving", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 661-676.

[64] A. Lakhotia and J.C. Deprez, "Restructuring programs by tucking statements into functions", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 677-691.

[65] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph-based program slicing", *IEEE Transactions on Software Engineering*, vol. 23, no. 4, 1997, pp. 246-259

[66] L.D. Larsen and M.J. Harrold, "Slicing object-oriented software", *Proceedings of 18th International Conference on Software Engineering*, Berlin, Germany, 1996, pp. 495-505.

[67] D. Liang and M.J. Harrold, "Reuse-driven interprocedural slicing in the presence of pointers and recursion", *Proceedings of International Conference on Software Maintenance*, Oxford, UK, 1999, pp. 421-432.

[68] P.E. Livadas and A. Rosenstein, "Slicing in the presence of pointer variables", Technical Report SERC-TR-74-F, Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994.

[69] J.R. Lyle and D. Binkley, "Program slicing in the presence of pointers", *Proceedings of Foundations of Software Engineering*, 1993.

[70] J.R. Lyle and M. Weiser, "Automatic program bug location by program slicing", *Proceedings of 2nd International Conference on Computers and Applications*, Peking, China, 1987, pp. 877-882.

[71] J.Q. Ning, A. Engberts, and W. Kozaczynski, "Recovering reusable components from legacy systems by program segmentation", *Proceedings of 1st Working Conference on Reverse Engineering*, Baltimore, Maryland, U.S.A., IEEE CS Press, 1993, pp. 64-72.

[72] A. Orso, S. Sinha, and M.J. Harrold, "Effects of pointers on data dependences and program slicing", Technical Reports GIT-CC-00-33, Georgia Institute of Technology, November 2000.

[73] Linda M. Ott, "Using slice profiles and metrics during software maintenance", *Proceedings of the 10th Annual Software Reliability Symposium*, 1992, pp. 16-23.

[74] L.M. Ott and J.M. Bieman, "Program slices as an abstraction for cohesion measurement", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 691-700.

[75] L. Ott and J. Thuss, "The relationship between slices and module cohesion", *Proceedings of the 11th International Conference on Software Engineering*, IEEE CS Press, 1989, pp. 198-204.

[76] K.J. Ottenstain and L.M. Ottenstain, "The program dependence graph in a software development environment", *ACM SIGPLAN Notices*, vol. 19, no. 5, 1984, pp. 177-184.

[77] T. Reps, "Program analysis via graph reachability", *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp. 701-726.

[78] F. Ricca and P. Tonella, "Web Application Slicing", *Proceedings of International Conference on Software Maintenance*, Florence, Italy, November 2001, IEEE CS Press (to appear).

[79] T. Richner and S. Ducasse, "Recovering high level views of object-oriented applications from static and dynamic information", *Proceedings of International Conference on Software Maintenance*, Oxford, UK, IEEE CS Press, 1999, pp. 13-22.

[80] S. Sinha, M.J. Harrold, and G. Rothermel, "System dependence-graph based slicing programs with arbitrary interprocedural control flow", *Proceedings of 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999, pp. 432-441.

[81] F. Tip, "A survey of program slicing techniques", *Journal of Programming Language*, vol. 3, 1995, pp. 121-189.

[82] F. Tip, J.D. Choi, J. Field, G. Ramalingham, "Slicing class hierarchies in C++", *Proceedings of 11th*

Conference on Object-Oriented Programming, Systems and Applications, San José, 1996, pp. 179-197.

[83] G.A. Venkatesh, "The semantic approach to program slicing", *ACM SIGPLAN Notices*, vol. 26, no. 6, 1991, pp. 107-119.

[84] M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method", PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[85] M. Weiser, "Programmers use slices when debugging", *Communications of the ACM*, vol. 25, 1982, pp. 446-452.

[86] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, 1984, pp. 352-357.

[87] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code", *Proceedings of Conference on Software Maintenance*, Orlando, FL, USA, IEEE CS Press, 1992, pp. 200-205.

[88] N. Wilde and M.C. Scully, "Software reconnaissance: mapping program features to code", *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, 1995, pp. 49-62.

[89] J. Zhao, J. Cheng, and K. Ushijima, "Static slicing of concurrent object-oriented programs", *Proceedings of 20th International Conference on Computer Software and Applications*, Seoul, Korea, IEEE CS Press, 1996, pp. 312-320.