RETROSPECTIVE:

Interprocedural Slicing Using Dependence Graphs

Susan Horwitz
Comp. Sci. Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
horwitz@cs.wisc.edu

Thomas Reps
Comp. Sci. Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
reps@cs.wisc.edu

David Binkley
Comp. Sci. Department
Loyola College
4501 North Charles Street
Baltimore, MD 21210
binkley@cs.loyola.edu

This paper made two main contributions: it defined *system dependence graphs* (SDGs), which extended program dependence graphs (PDGs) [31, 17] to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. It then defined an interprocedural slicing algorithm that identifies the components of the SDG that might affect the values of the variables defined at or used at a given program point p, and/or how often p executes. The novelty of the algorithm was that it correctly accounts for the calling context of a called procedure, thereby improving on the technique for interprocedural slicing algorithm that had been presented in Mark Weiser's seminal papers on program slicing [47, 48].

The paper's results form part of David Binkley's Ph.D. thesis [6], and a journal version of the paper was published in TOPLAS [20]. SDGs are covered by a U.S. patent [40]; algorithms for forward and backward slicing were part of the original patent application that was submitted, but the Wisconsin Alumni Research Foundation decided that enough money had been spent by the time the patent office was convinced that Claim 1 (SDGs) was patentable.

The data structures and slicing algorithms described in the paper are used in one commercial product, CodeSurfer®, a tool for code understanding and inspection that supports browsing and slicing of dependence graphs. CodeSurfer is available from GrammaTech, Inc. (see http://www.grammatech.com/products/codesurfer/). (The slicing algorithm used in CodeSurfer is actually based on the improved method described in [23].)

The PLDI 88 paper was careful to state what slicing problem it did *not* solve, i.e., that of producing a slice that is an executable projection of the original program. The reason has to do with multiple calls to the same procedure: in general, a slice may include more than one such call, and at each, a different subset of the procedure's parameters may be in the slice. An extended slicing algorithm that addresses this parameter-mismatch problem was later given by Binkley [7].

The PLDI 88 paper was couched in terms of a very simple programming language. However, one can think of SDGs as a *class* of program representations: to represent programs in different programming languages, one uses different variants of PDGs containing nodes and edges that capture the features and constructs of the language. The issue of how to create appropriate PDGs/SDGs is really orthogonal to the issue of how to slice them. There has been considerable work on how to build dependence graphs for the features and constructs found in real-world programming languages, including arrays [4, 50, 29, 18, 32], reference parameters [20], pointers [28, 19, 10], non-structured control flow [2, 11, 1, 46, 27], and threads [26, 15, 30].

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

Context-Sensitive Program Analysis

Nowadays the interprocedural-slicing method of the PLDI 88 paper would be termed a "context-sensitive" interprocedural-slicing method. In general, a context-sensitive analysis is one in which the analysis of a called procedure is "sensitive" to the context in which it is called. A context-sensitive analysis captures the fact that calls to a procedure through different call sites can have different effects on the program's execution state. General frameworks for context-sensitive program analysis already existed in 1988 [13, 45], although we were not aware of them at the time. Our work was more influenced by Kastens's algorithm for building a collection of cooperating finite-state machines for evaluating derivation trees of attribute grammars [24] (see below).

The two other papers in the session at PLDI 88 in which our paper was presented [8, 12] actually foreshadowed a major direction that our work was to take subsequently: Callahan presented a context-sensitive algorithm for a certain kind of dataflow-analysis problem [8]; Cooper and Kennedy presented a graph-reachability algorithm for finding procedures' MayMod and MayUse sets (which summarize possible side-effects of procedure calls on global variables) [12]. Later work by Horwitz, Reps, and collaborators showed how many dataflow-analysis problems similar to the one that Callahan addressed could be turned into graph-reachability problems in a fashion similar to the technique used in the Cooper-Kennedy paper—and then solved using algorithms very similar to the Horwitz-Reps-Binkley interprocedural-slicing algorithm. However, even though the three papers were presented in the same hour and a half session at PLDI 88, the act of combining their techniques did not occur for another six years.

Context-Free-Language Reachability

The PLDI 88 paper pointed out a correspondence between the call structure of a program and a context-free grammar, and between the *intra*procedural transitive dependences among a PDG's parameter nodes and the dependences among attributes in an attribute grammar [25]. Our paper exploited this correspondence to compute summary edges that were added to the SDG at call sites to capture certain transitive dependences. The transitive dependences of interest are very much like the transitive dependences captured by the characteristic graphs of an attribute grammar's nonterminals; the algorithm for identifying summary edges was inspired by the first two passes of the algorithm for creating the "TDS graphs" that are used to define the class of ordered attribute grammars [24]. This

yields a slicing algorithm that is quartic (i.e., $O(N^4)$) in a certain size parameter of the problem.

Given what we have learned in the last fifteen years, if we were to write the paper today, we would describe the work somewhat differently: in addition to describing the problem as a context-sensitive analysis, we would say that the algorithm for identifying summary edges was a tabulation algorithm (or a dynamic-programming algorithm); we might explain how interprocedural slicing can be formulated in terms of a simple logic program, along the lines of [35]; and we would explain how the interprocedural-slicing problem is an example of the following kind of generalized graph-reachability problem in which labels on the edges of the graph are used to filter out paths that are not of interest:

> A path P from vertex s to vertex t only counts as a "valid connection" between s and t if the word spelled out by P (i.e., the concatenation, in order, of the labels on the edges of P) is in a certain language.

When the filter language is a context-free language, this is called CFL-reachability, a concept originated by Yannakakis in 1990 [51]. CFL-reachability problems can be solved in time cubic in the number of nodes in the graph (and, in general, no algorithm with a better running time is known). For a survey of ways in which CFL-reachability applies to program-analysis problems, see [38].

(A slight clash in terminology exists. In formal-language theory, the "context-sensitive languages" are a strictly more powerful formalism than the context-free languages. Unfortunately, the principle that context-*free*-language reachability captures an essential aspect of context-*sensitive* analysis was fully articulated [38] only after the term "context-sensitive analysis" had been adopted by the programming-languages community [16, 49].)

In the case of interprocedural slicing (and in many other context-sensitive program-analysis problems) the filter language is a language of matched parentheses (also called a Dyck language). In essence, the PLDI 88 paper used a language of partially balanced parentheses to exclude from consideration paths in which calls and returns are mismatched. The parentheses are defined as follows [23]: Let each call-site node in SDG G be given a unique index from 1 to TotalCallSites, where TotalCallSites is the total number of call sites in the program. For each call site c_i, label the outgoing parameter-in edges and the incoming parameter-out edges with the symbols "$(_i$" and "$)_i$", respectively; label the outgoing call edge with "$(_i$". Label all other edges in G with the symbol e.

Dyck languages had been used in earlier work on interprocedural dataflow analysis by Sharir and Pnueli to specify that the contributions of certain kinds of infeasible execution paths should be filtered out [45]; however, the dataflow-analysis algorithms given by Sharir and Pnueli are not based on graph reachability.

As mentioned above, the algorithm given in the PLDI 88 paper is of quartic time complexity. However, Dyck languages are context-free, which means that Yannakakis's result applies. Although Yannakakis's work means that all CFL-reachability problems can be solved in time cubic in the number of graph nodes, one can sometimes do asymptotically better than this by taking advantage of the structure of the graph that arises in a particular subclass of problems. For instance, an improved algorithm for interprocedural slicing is described in [23]; this algorithm is cubic in a certain size parameter, but this parameter is usually smaller than the size of the SDG's node set.

Dyck-language reachability was shown by Reps, Sagiv, and Horwitz to provide algorithms for a wide variety of interprocedural program-analysis problems [43], including the one that was addressed in Callahan's PLDI 88 paper [8]. These ideas were elaborated on in a sequence of papers that showed the utility of the approach for both exhaustive and demand program-analysis problems [23, 41, 21, 22]. These algorithms all have cubic worst-case running time (and use quadratic space), although for some interesting special cases, such as gen-kill dataflow-analysis problems, they run in linear time and use linear space.

Although the authors became aware of the connection to the more general concept of CFL-reachability sometime in the fall of 1994, of the papers from this period, only [36] mentions CFL-reachability explicitly and references Yannakakis's paper [51].

Sagiv, Reps, and Horwitz used similar ideas—but went beyond just CFL-reachability—to give more powerful, but still cubic-time, interprocedural dataflow-analysis algorithms [44]. Among other benefits, this work showed how to obtain precise jump functions for certain classes of interprocedural constant-propagation problems, thereby improving on those developed by Callahan et al. in their CC 86 paper [9].

The CFL-reachability viewpoint has also yielded results on inherent computational limitations related to slicing and other program-analysis problems [37, 39, 33]. Other work that has adopted the CFL-reachability model has addressed such topics as program chopping [42], type checking [34], model checking [3, 5], and bug detection [14].

REFERENCES

[1] H. Agrawal. On slicing programs with jump statements. In *Conf. on Prog. Lang. Design and Impl.*, pages 302–312, 1994.

[2] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Int. Workshop on Automated and Algorithmic Debugging*, pages 206–222, 1993.

[3] T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Workshop on Prog. Analysis for Softw. Tools and Eng.*, New York, NY, June 2001. ACM Press.

[4] U. Bannerjee. *Speedup of Ordinary Programs*. PhD thesis, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL, October 1979.

[5] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP '01*, 2001.

[6] D. Binkley. *Multi-Procedure Program Integration*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, August 1991. Tech. Rep. TR-1038.

[7] D. Binkley. Precise executable interprocedural slices. *Let. on Prog. Lang. and Syst.*, 2:31–45, 1993.

[8] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Conf. on Prog. Lang. Design and Impl.*, pages 47–56, New York, NY, 1988. ACM Press.

[9] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Symp. on Comp. Construct.*, pages 152–161, 1986.

[10] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[11] J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *Trans. on Prog. Lang. and Syst.*, 16(4):1096–1113, 1994.

[12] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Conf. on Prog. Lang. Design and Impl.*, pages 57–66, New York, NY, 1988. ACM Press.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor,

Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977), pages 237–277. North-Holland, 1978.

[14] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Conf. on Prog. Lang. Design and Impl.*, pages 57–68, New York, NY, 2002. ACM Press.

[15] M.B. Dwyer, J.C. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng. Slicing multi-threaded Java programs: A case study. Tech. Rep. 99-7, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, KS, February 1999.

[16] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conf. on Prog. Lang. Design and Impl.*, New York, NY, 1994. ACM Press.

[17] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.

[18] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Conf. on Prog. Lang. Design and Impl.*, pages 15–29, New York, NY, 1991. ACM Press.

[19] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.

[21] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Symp. on the Found. of Softw. Eng.*, pages 104–115, New York, NY, October 1995. ACM Press.

[22] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. report TR-1283, Comp. Sci. Dept., Univ. of Wisconsin, August 1995. Available at "http://www.cs.wisc.edu/wpis/papers/tr1283r.ps".

[23] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Symp. on the Found. of Softw. Eng.*, pages 11–20, New York, NY, December 1994. ACM Press.

[24] U. Kastens. Ordered attribute grammars. *Acta Inf.*, 13(3):229–256, 1980.

[25] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968.

[26] J. Krinke. Static slicing of threaded programs. In *Workshop on Prog. Analysis for Softw. Tools and Eng.*, June 1998.

[27] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Colloq on Formal Approaches in Softw. Eng.*, pages 96–112, 2002.

[28] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.

[29] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *Conf. on Prog. Lang. Design and Impl.*, pages 1–14, New York, NY, 1991. ACM Press.

[30] M. Nanda and S. Ramesh. Slicing concurrent programs. In *Symp. on Softw. Testing and Analysis*, August 2000.

[31] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Softw. Eng. Symp. on Practical Softw. Dev. Environments*, pages 177–184, New York, NY, 1984. ACM Press.

[32] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Conf. on Prog. Lang. Design and Impl.*, pages 140–151, New York, NY, 1992. ACM Press.

[33] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *Trans. on Prog. Lang. and Syst.*, 22(2):416–430, 2000.

[34] J. Rehof and M. Fähndrich. Type-base flow analysis: From polymorphic subtyping to CFL-reachability. In *Symp. on Princ. of Prog. Lang.*, pages 54–66, New York, NY, 2001. ACM Press.

[35] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic Publishers, 1994.

[36] T. Reps. Shape analysis as a generalized path problem. In *Symp. on Part. Eval. and Semantics-Based Prog. Manip.*, pages 1–11, New York, NY, June 1995. ACM Press.

[37] T. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Inf.*, 33:739–757, 1996.

[38] T. Reps. Program analysis via graph reachability. *Inf. and Softw. Tech.*, 40(11-12):701–726, November 1998.

[39] T. Reps. Undecidability of context-sensitive data-dependence analysis. *Trans. on Prog. Lang. and Syst.*, 22(1):162–186, January 2000.

[40] T. Reps, S. Horwitz, and D. Binkley. U.S. Patent Number 5,161,216, Interprocedural slicing of computer programs using dependence graphs, November 1992.

[41] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM Press.

[42] T. Reps and G. Rosay. Precise interprocedural chopping. In *Symp. on the Found. of Softw. Eng.*, New York, NY, October 1995. ACM Press.

[43] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Tech. Rep. TR 94-14, Datalogisk Institut, Univ. of Copenhagen, 1994. Available at "http://www.cs.wisc.edu/wpis/papers/diku-tr94-14.ps".

[44] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.

[45] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[46] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary control flow. In *Int. Conf. on Softw. Eng.*, pages 432–441, Wash., DC, 1999. IEEE Comp. Soc.

[47] M. Weiser. Program slicing. In *Int. Conf. on Softw. Eng.*, pages 439–449, Wash., DC, 1981. IEEE Comp. Soc.

[48] M. Weiser. Program slicing. *Trans. on Softw. Eng.*, SE-10(4):352–357, July 1984.

[49] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conf. on Prog. Lang. Design and Impl.*, pages 1–12, New York, NY, 1995. ACM Press.

[50] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL, October 1982.

[51] M. Yannakakis. Graph-theoretic methods in database theory. In *Symp. on Princ. of Database Syst.*, pages 230–242, 1990.