Advanced Slicing of Sequential and Concurrent Programs

Jens Krinke
FernUniversität in Hagen, Germany*
Jens.Krinke@FernUni-Hagen.de

Abstract

Program slicing is a technique to identify statements that may influence the computations in other statements. Despite the ongoing research of almost 25 years, program slicing still has problems that prevent a widespread use: Sometimes, slices are too big to understand and too expensive and complicated to be computed for real-life programs. The presented thesis shows solutions to these problems: It contains various approaches which help the user to understand a slice more easily by making it more focused on the user's problem. All of these approaches have been implemented in the VALSOFT system and thorough evaluations of the proposed algorithms are presented.

The underlying data structures used for slicing are program dependence graphs. They can also be used for different purposes: A new approach to clone detection based on identifying similar subgraphs in program dependence graphs is presented; it is able to detect modified clones better than other tools.

In the theoretical part, this thesis presents a high-precision approach to slice concurrent procedural programs despite the fact that optimal slicing is known to be undecidable. It is the first approach to slice concurrent programs that does not rely on inlining of called procedures.

1. Introduction

Program slicing answers the question "Which statements may affect the computation at a different statement?", something every programmer asks once in a while. After Weiser's first publication on slicing in 1979, almost 25 years have passed and various approaches to compute slices have evolved. Usually, inventions in computer science are adopted widely after around 10 years. Why are slicing techniques not easily available yet? William Griswold gave a talk at PASTE 2001 [4] on that topic: *Making Slicing Practical: The Final Mile*. He pointed out why slicing is still not widely used today. The two main problems are:

1. Available slicers are slow and imprecise.
2. Slicing 'as-it-stands' is inadequate to essential software-engineering needs.

Not everybody agrees with his opinion. However, his first argument is based on the observation that research has generated fast and precise approaches but scaling the algorithms for real-world programs with million lines of code is still an issue. Precision of slicers for sequential imperative languages has reached a high level, but it is still a challenge for the analysis of concurrent programs—only lately is slicing done for languages with explicit concurrency like Ada or Java. The second argument is still valid: Usually, slices are hard to understand. This is partly due to bad user interfaces, but is mainly related to the problem that slicing 'dumps' the results onto the user without any explanation.

The thesis [11] presented here tries to show how these problems and challenges can be tackled. Therefore, the three main topics are:

1. Present ways to slice concurrent programs more precisely.
2. Help the user to understand a slice more easily by making it more focused on the user's problem.
3. Give indications of the problems and consequences of slicing algorithms for future developers.

Furthermore, this thesis gives a self-contained introduction to program slicing. It does not try to give a complete survey because since Tip's excellent survey [21][1] the literature relevant to slicing has exploded: CiteSeer recently reported 257 citations of Weiser's slicing article [24] (and 95 for [23]). This thesis only contains 187 references where at least 108 have been published after Tip's survey.

*The work presented in this thesis was mainly performed at the Universität Passau, Germany. The complete thesis is available at http://www.fernuni-hagen.de/ST/diss.php

[1] Tip's survey [21] has been followed by some others [1, 5, 2, 6].

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

2. Slicing

A slice extracts those statements from a program that potentially have an influence on a specific statement of interest, which is the slicing criterion. Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [22, 24]. The other main approach to slicing uses reachability analysis in program dependence graphs [3]. Program dependence graphs mainly consist of nodes representing the statements of a program as well as control and data dependence edges:

- Control dependence between two statement nodes exists if one statement controls the execution of the other (e.g. at if or while statements).
- Data dependence between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.

A slice can now be computed simply in three steps: Map the slicing criterion on a node, find all backward reachable nodes, and map the reached nodes back on the statements.

Slicing has found its way into various applications. Nowadays it is probably mostly used in the area of software maintenance and reengineering. Specifically, applications are Debugging, Testing, Program Differencing and Integration, Impact Analysis, Function Extraction and Restructuring, or Cohesion Measurement. It has even been used for debugging and testing spreadsheets or type checking programs.

2.1. Slicing Sequential Programs

Example 1 (Slicing without Procedures) *Figure 1 shows a first example where a program without procedures shall be sliced. To compute the slice for the statement* `print a`*, we just have to follow the shown dependences backwards. This example contains two data dependences and the slice includes the assignment to* `a` *and the read statement for* `b`*.*

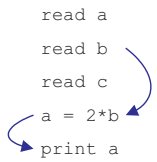

Figure 1. A procedure-less program

In all examples, we will ignore control dependence and just focus on data dependence for simplicity of presentation. Also, we will always slice backwards from the `print a` statement.

Slicing without procedures is trivial: Just find reachable nodes in the PDG [3]. The underlying assumption is that all paths are *realizable*. This means that a possible execution of the program exists for any path that executes the statements in the same order.

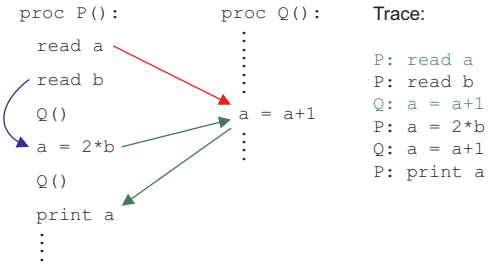

Figure 2. A program with two procedures

Example 2 (Slicing with Procedures) *Now, the example is extended by adding procedures in Figure 2. If we ignore the calling context and just do a traversal of the data dependences, we would add the* `read a` *statement into the slice for* `print a`*. This is wrong because this statement clearly has no influence on the* `print a` *statement. The* `read a` *statement only has an influence on the first call of procedure* `Q` *but* `a` *is redefined before the second call to procedure* `Q` *through the assignment* `a=2*b` *in procedure* `P`*.*

Such an analysis is called *context-insensitive* because the calling context is ignored. Paths are now considered realizable only if they obey the calling context. Thus, slicing is *context-sensitive* if only realizable paths are traversed. Context-sensitive slicing is solvable efficiently—one has to generate summary edges at call sites [7]: Summary edges represent the transitive dependences of called procedures at call sites.

Within the implemented infrastructure to compute PDGs for ANSI C programs, various slicing algorithms have been implemented and evaluated. One of the evaluations of this thesis (presented in [10]) shows that context-insensitive slicing is very imprecise in comparison to context-sensitive slicing. On average, slices computed by the context-insensitive algorithm are 67% larger than the ones computed by the context-sensitive algorithm. This shows that context-sensitive slicing is highly preferable because the loss of precision is not acceptable. A surprising result is that the simple context-insensitive slicing is *slower* than the more complex context-sensitive slicing (23% on average). The reason is that the context-sensitive algorithm has to visit many fewer nodes during traversal due to its higher precision. Both algorithms usually visit a node or an edge only once, the context-sensitive algorithm has to visit a few nodes twice.

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

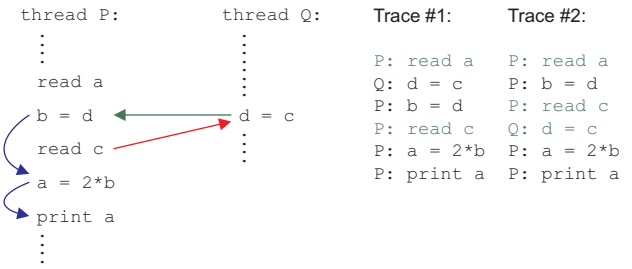

Figure 3. A program with two threads

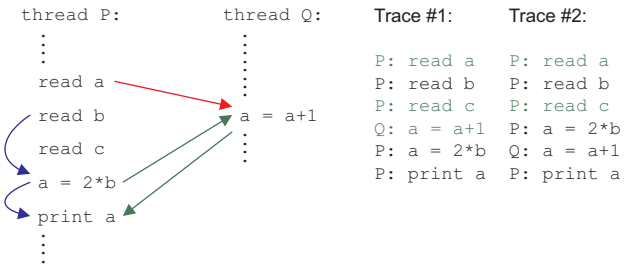

Figure 4. Another program with two threads

2.2. Slicing Concurrent Programs

Now, let's move on to concurrent programs. In concurrent programs that share variables, another type of dependence arises: *interference*. Interference occurs when a variable is defined in one thread and used in a concurrently executing thread.

Example 3 (Slicing Concurrent Programs) *In the example in Figure 3 we have two threads* `P` *and* `Q` *that execute in parallel. In this example, there are two interference dependences: One is due to a definition and a usage of variable* `d`*, the other is due to accesses to variable* `c`*.*

A simple traversal of interference during slicing will make the slice imprecise because interference may lead to unrealizable paths again. In the example in Figure 3, a simple traversal will include the `read c` statement into the slice. But there is no possible execution where the `read c` statement has an influence on the assignment `b=d`. A matching execution would require *time travel* because the assignment `b=d` is always executed before the `read c` statement. A path through multiple threads is now realizable if it contains a valid execution chronology. However, even when only realizable paths are considered, the slice will not be as precise as possible. The reason for this imprecision is that concurrently executing threads may *kill* definitions of other threads.

Example 4 *In the example in Figure 4, the* `read a` *statement is reachable from the* `print a` *statement via a realizable path. But there is no possible execution where the* `read` *statement has an influence on the* `print` *statement when assuming that statements are atomic. Either the* `read` *statement reaches the usage in thread* `Q` *but is redefined afterwards through the assignment* `a=2*b` *in thread* `P`*, or the* `read` *statement is immediately redefined by the assignment* `a=2*b` *before it can reach the usage in thread* `Q`*.*

Müller-Olm has shown that precise context-sensitive slicing of concurrent programs is undecidable in general [16]. Therefore, we have to use conservative approximations to analyze concurrent programs. A naive approximation would allow time travel, causing an unacceptable loss of precision. Also, we cannot use summary edges to be context-sensitive because summary edges would *ignore* the effects of parallel executing threads. Summary edges represent the transitive dependences of the called procedure without interference; they cannot be extended to represent interference, because interference is not transitive. Again, reverting to context-insensitive slicing would cause an unacceptable loss of precision.

To be able to provide precise slicing without summary edges, new slicing algorithms have been developed based on capturing the calling context through *call strings* [18]. Call strings can be seen as a representation of call stacks. They are frequently used for context-sensitive program analysis, e.g. pointer analysis. The call strings are propagated along the edges of the PDG: At edges that connect procedures, the call string is used to check that a call always returns to the right call site. Thus, call strings are never propagated along unrealizable paths.

The basic idea for the high-precision approach to slice concurrent programs is the adaption of the call string approach to concurrent programs. The context is now captured through one call string for each thread. It is then a tuple of call strings which is propagated along the edges in PDGs. On the one hand, they enforce that propagation returns to the right call site from a called procedure, and on the other hand, they ensure that no time travel occurs during the traversal between threads.

A combined approach avoids combinatorial explosion of call strings: Summary edges are used to compute the slice within threads. Additionally, call strings are only generated and propagated along interference edges if the slice crosses threads. With this approach many fewer contexts are propagated.

This only outlines the idea of the approach—this thesis presents the foundations and algorithms for slicing sequential and concurrent programs in detail (also presented in [12]). Additionally, a major part of this thesis presents optimizations and advanced applications of slicing.

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

3. Contributions

This thesis is self-contained as much as possible. Besides a thorough presentation of slicing, the accomplishments of this thesis are:

- A fine-grained program dependence graph which is able to represent ANSI C programs including non-deterministic execution order. It is a self-contained intermediate representation and the base of clone detection and path condition computation.

- A high-precision approach to slicing concurrent procedure-less programs. A preliminary version has been published as [8].

- A new approach to slicing concurrent procedural programs. This context-sensitive approach reaches a high precision, despite the fact that precise or optimal slicing is undecidable. This is the first approach that does not need inlining and is able to slice concurrent recursive programs (published as [12]).

- Some variations of slicing and chopping algorithms within interprocedural program dependence graphs and a thorough evaluation of these algorithms. Most of this has already been published in [10]. These algorithms include call string based variants, which are needed for slicing concurrent programs.

- Fundamental ideas and approaches for visualizing dependence graphs and slices as graph based, textual and abstract representations [14]. Experience shows that the graphical presentation is less helpful than expected and a textual presentation is superior in most cases. Another, more sophisticated approach visualizes the influence range of chops for variables and procedures. This enables a visualization of the impact of procedures and variables on the complete system.

- Some methods to make the results of slicing more focused or more abstract. Parts of this have been published as [13] and presents an approach that can be used to 'filter' slices. It basically introduces 'barriers' which are not allowed to be passed during slice computation. The barrier variants of slicing and chopping provide filtering possibilities for smaller slices and better comprehensibility.

- Techniques to reduce the size of program dependence graphs without worsening the precision of slicing. This relies on elimination of redundant nodes and folding of strongly connected components.

- An approach to clone detection based on program dependence graphs. This approach has a higher detection rate for modified clones than other approaches, because it identifies similar semantics instead of similar texts. After publication in [9], the benefits and drawbacks of this approach have been evaluated in a clone detection contest.

- Methods to generate path conditions for complex data structures, procedures and concurrent programs. The general approach of path conditions was introduced by Snelting [19] and developed further by Robschink [17, 20]. Path conditions provide necessary conditions under which an influence between the source and target criterion exists.

- The design of the VALSOFT system and implementation of the data flow analysis, dependence graph construction and various slicing and chopping algorithms within it.

4. Conclusions

The presented thesis attacked important problems in program slicing and showed some solutions. With the newly developed slicing techniques and their evaluation it was possible to show that highly precise and efficient slicing is possible for sequential and concurrent programs. The implementation in the VALSOFT system exposed that the slicing itself is not responsible for scalability problems, but the data flow analyses, and pointer analysis in particular, needed to build the program dependence graphs.

In the original form, program slicing is not well suited for program comprehension. This thesis presented approaches with new slicing techniques that generate more comprehensible results. Methods that not only present results, but also explain them, show promising results.

Acknowledgments. I am grateful for all the support provided by my advisor Gregor Snelting. The former and current members of the software systems group in Passau deserve special thanks for their help and valuable discussions.

References

[1] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[2] A. De Lucia. Program slicing: Methods and applications. In *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001. Invited paper.

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

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

[4] W. G. Griswold. Making slicing practical: The final mile, 2001. Invited Talk, PASTE'01.
[5] M. Harman and K. B. Gallagher. Program slicing. *Information and Software Technology*, 40(11–12):577–581, 1998.
[6] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
[7] S. B. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.
[8] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42. ACM Press, 1998. ACM SIGPLAN Notices 33(7).
[9] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.
[10] J. Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, pages 22–31, 2002.
[11] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.
[12] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings ESEC/FSE*, pages 178–187, 2003.
[13] J. Krinke. Slicing, chopping, and path conditions with barriers. *Software Quality Journal*, 12(4), 2004.
[14] J. Krinke. Visualization of program dependence and slices. In *International Conference on Software Maintenance*, 2004.
[15] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12):661–675, Dec. 1998.
[16] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC 2001 (33th ACM Symposium on Theory of Computing)*, pages 647–656, 2001.
[17] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proceedings of the 24th International Conference of Software Engineering (ICSE)*, pages 478–488, 2002.
[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
[19] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348. Springer, 1996.
[20] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. Submitted for publication, 2003.
[21] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3), Sept. 1995.
[22] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.
[23] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
[24] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**