Towards Probabilistic Program Slicing

Jeremy Singer
School of Computer Science
University of Manchester
jsinger@cs.man.ac.uk

May 2006

Abstract

This paper outlines the concept of *probabilistic* program slicing. Whereas conventional slicing removes statements that cannot affect the slicing criterion, probabilistic slicing also removes statements that are unlikely to affect the criterion. The paper presents a simple example before describing some algorithmic concerns. Then three motivating applications are described. Finally it highlights existing work that may be built upon, and future work that needs immediate attention if this idea is to succeed.

1 Introduction

There is an increasing trend of applying probabilistic extensions to classical program analysis techniques. Successful examples include abstract interpretation [PW00, PHW05], points-to analysis [CHH$^+$03], bit-width analysis [ÖNG04] and model checking [KNP04, KNP05]. In general, two motivations are given for probabilistic extensions.

1. to analyse programs that exhibit stochastic behaviour.
2. to derive probabilistic properties for classical programs.

This paper is an initial attempt to formulate a probabilistic version of *program slicing* [Wei81]. It shows how to incorporate probability information into a standard slicing algorithm. Then it discusses why probability information may be useful in certain applications of slicing. Throughout this paper, we restrict consideration to classical (non-stochastic) programs, for which we would like to construct *probabilistic slices*, i.e. program slices that have a given probability of being correct.

Dagstuhl Seminar Proceedings 05451
Beyond Program Slicing
http://drops.dagstuhl.de/opus/volltexte/2006/485

```
1:  integer x, y, z, n
2:  n ← input()
3:  y ← 0
4:  z ← −1
5:  if (prime(n))
6:      then if (odd(n))
7:          then x ← 3
8:          else x ← 2 + z
9:      else x ← 1
10: return x
```

Figure 1: Simple example program

source line	*test*	*# executions*	*# then's*	*# else's*
5	prime(n)	1000	168	832
6	odd(n)	168	167	1

Figure 2: Probabilities of conditional branch outcomes

1.1 Simple example

Figure 1 shows a simple example program[1] which will motivate probabilistic slicing. A standard syntax-preserving static backward slice on the function return value, variable x, would only remove the assignment to unused variable y. Now suppose that the value of the program input n is uniformly distributed over numbers in the range $[2, 1001]$. (Note that this assumption is implicit for the remainder of this section.) Then it is clear to see that the **then** and **else** branches of the **if** statements are skewed. In the outer conditional test (line 5), most values of n will not be prime. In the inner conditional test (line 6), most prime values of n will be odd. This information can be incorporated into the analysis by associating probability information with each **if** statement. Figure 2 shows the branch outcome frequencies, assuming n takes a different value in the range $[2, 1001]$ over all 1000 executions.

The aim of conventional program slicing is to remove *irrelevant* statements from the program, with respect to the slicing criterion. This relies on the well-known notion of static dependence (see Section 2.1). In general, static program slices are conservative with respect to safety, they consider dependences that can occur over all possible paths through the program.

[1] This example is reworked from notes by Chris Hankin et al at `http://www.doc.ic.ac.uk/~herbert/epsrc/node5.html`.

source lines in minimal end-slice	*probability of correctness*
$\{\}$	0.000
$\{1, 9, 10\}$	0.832
$\{1, 2, 5, 7, 9, 10\}$	0.999
$\{1, 2, 4, 5, 6, 7, 8, 9, 10\}$	1.000

Figure 3: Various probabilistic slices of example program

When we incorporate probability information, it will be possible to remove *infrequent* statements as well as irrelevant statements. This provides a more dynamic notion of dependence. Probabilistic slices are not conservative since they only take into account dependences from a certain proportion of program paths, rather than all program paths.

Suppose we restrict consideration to conditional branch outcomes that have a probability of more than 0.8. (If we encounter conditional tests that are less skewed than this, then we must consider both outcomes as potentially belonging to the slice.)

The conditional test at line 5 has an **else** probability of 0.832, so we ignore the infrequent **then** branch including the nested conditional test.

Then the end-slice consists of the final assignment to x on line 9, supplemented by the scaffolding of lines 1 and 10. Of course, now the slice is not guaranteed to be correct! It is correct with probability 0.832. Figure 3 shows the different statement-minimal end-slices [Dan99] of the example program, together with their probability of correctness. Note that when the correctness probability is set to 1, then the probabilistic slice is equivalent to a conventional, non-probabilistic slice.

1.2 Contributions

This paper makes three significant contributions.

1. Section 2 outlines an initial algorithm for probabilistic program slicing. This is a simple intraprocedural formulation for a basic `while` language. Extensions will be necessary for real-world programming languages.

2. Section 3 provides some motivating examples for probabilistic slicing. There may be many more applications, but these early suggestions were made by participants at the Dagstuhl seminar 05451 on 'Beyond Program Slicing'.

3. Sections 4 and 5 discuss related work and future work respectively. These show the small amount of existing work in the field that can be used as a foundation, and point out challenges that need to be met if probabilistic slicing is to gain widespread acceptance.

2 Algorithmic Details

Program analyses are required to obtain both dependence and probability information. Initially we aim to base our probabilistic slicing algorithm on simple syntax-preserving static backward slicing.

2.1 Dependence Computation

Dependences are obtained from a slicing algorithm. We propose to use a standard slicing algorithm based on the control flow graph (CFG) [Wei81] or program dependence graph (PDG) [OO84]. This style of algorithm will compute both data and control dependences between statements. Recall that statement X is *data dependent* on statement Y if changing the values of variables defined at Y may affect the values of variables used at X. Recall that statement X is *control dependent* on statement Y if the outcome of Y affects whether or not X is executed. In general, syntax-preserving static backward slicing involves computing transitive closure of these dependence edges from the slicing criterion.

2.2 Probability Computation

Section 1 assumed that the input values for the example program were uniformly distributed over a fixed range of integers. This was a simplification for the purposes of presentation. In general, more sophisticated mechanisms are required to acquire probability information.

There are three main approaches to calculate these conditional branch probabilities.

1. The simplest method is *static* estimation of branch probabilities. Hennessy and Patterson [HP03] outline this approach for static branch prediction. For instance, we could assume that an **if** statement is equally likely to go either way, and that a **while** condition has a 90% chance of repeating the loop and a 10% chance of terminating the loop. More sophisticated static analyses are possible. For instance Clark et al [CHM05] provide rules for how different program statements transform probability distributions of program inputs, although their work is in the field of information flow security.

2. A more complicated method is *quasi-dynamic* analysis, also known as feedback-directed analysis. The program is run on a test set of input data, and the observed conditional branch outcomes are used to calculate their probabilities. These probabilities are assumed to hold good for all subsequent input data.

3. The best method is *dynamic analysis* in an adaptive runtime infrastructure such as Jikes RVM [AAB+00, AAB+05]. In this way, branch probabilities are constantly updated for input data that varies over time. This method enables online probabilistic program slicing.

2.3 Combining Dependence and Probability

This section discusses the method for combining both dependence and probability information in order to construct a reduced version of the original program (a probabilistic slice). Recall that dependence information is used to remove *irrelevant* statements, and probability information is used to remove *infrequent* statements.

2.3.1 Restrictions

In this paper we will only formulate probabilistic slicing for a simple intraprocedural while language, consisting of **if** statements and **while** loops. We will allow standard integer arithmetic operations, assignments to scalar variables, standard comparison operations, also some built-in predicates like odd() and prime() as we saw in the example program in Figure 1.

We assume that all loops eventually terminate. Probability information is only attached to branches of **if** statements, not to branches associated with **while** loops.

A probabilistic slicing criterion (V, p, t) will consist of a set of variables V, a program point p and a probability threshold t. There are three different ways to use this information to conduct probabilistic slicing.

2.3.2 Apply Probability Information Before Dependence Information

In this scheme, we first reduce the program by removing infrequently executed statements. Statements are deemed to be infrequent if they are situated at program points that have a probability of being executed of less than the threshold value t. These program point probabilities are easily calculated from the conditional branch probabilities using the elementary probability theory outlined below.

Given a while program w to analyse, transform w into control flow graph (CFG) form [ASU86, App98], where nodes represent basic blocks of consecutive program statements and edges represent the possible flow of control between statements. This can be a simple syntax-directed translation from source code to CFG. Figure 4 shows the CFG for the example while program from Figure 1. Annotate the outgoing edges of **if** statements with the conditional branch probabilities, as computed using one of the approaches from Section 2.2. Only the edges corresponding to **then** and **else** constructs have probability annotations. All other edges are unconditional (even edges in loop tests, since we do not handle loops properly yet).

So, each **then**/**else** edge pair has a probability distribution. It should always be the case that $P_{then} + P_{else} = 1$. This assumption holds in all three approaches for computing branch probabilities given in Section 2.2. Implicitly, this means that each probability is *conditioned* on the fact that the corresponding **if** statement is executed. Thus all the conditional branch probabilities (apart from those in the top-level **if** statements) are conditional. This makes program point

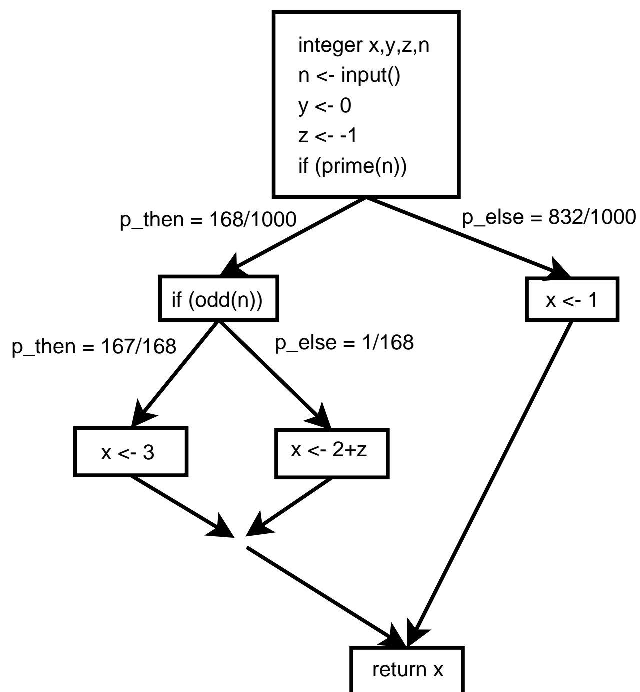

Figure 4: Control flow graph representation of example program, with probability annotations for then and else edges

probability calculation straightforward since it simply requires multiplication of the appropriate conditional probabilities.

To determine the probability that point p will be executed, find the probability-annotated edges (**then** and **else** edges) that *dominate*[2] p. (Recall that edge e dominates point p if every path from the program entry point to p includes e. Note that dominance is more usually a relation between CFG nodes, but it is also valid to apply it for edges [JP93].) Now the product of the annotated probabilities of the dominating edges will give the probability that p will be executed. This calculation should be applied to all statements (or more practically, once per basic block). The formula for the calculation is:

$$P(p \text{ is executed}) = \Pi_{e \in E} P(e \text{ is executed} \mid \text{corresponding } \textbf{if} \text{ stmt is executed})$$

where

$$E = \{e \mid e \in \text{set of edge-annotated edges} \wedge e \in \text{set of edges that dominate } p\}$$

Now all program points have been assigned execution probabilities, it is possible to remove infrequently executed statements. All statements whose execution probability is lower than the threshold value $(1-t)$ should be removed, since in this case, t is the probability that the slice is correct.

For example, consider the program point labelled 8 in the program in Figure 1. Point 8 is dominated by the **then** branch at point 6 and the **else** branch at point 8. The product of their conditional probabilities is $0.168 * 0.994 = 0.167$. So when the threshold value $(1-t) > 0.167$, this point should be deleted from the program.

Finally the reduced program is sliced according to the classical slicing criterion (V, p) using a standard static backward slicing algorithm. Of course, the empty slice is necessarily returned if the slicing criterion point p was removed in the first phase of this analysis!

2.3.3 Apply Dependence Information Before Probability Information

In this scheme, we first slice the program according to the slicing criterion (V, p) using a standard static backward slicing algorithm.

Next we calculate the probability that each statement belongs to the slice, using the conditional branch probabilities of the **if** statements contained in the slice. It is possible to use the same mechanisms as above to compute execution probabilities for all statements from conditional probabilities of **then/else** branches, using products of conditional probabilities. However, it is necessary to make adjustments for cases where one arm of a conditional is in the slice but the other arm has been sliced away! In such cases, we simply treat the retained arm as being unconditionally executed, so the probability annotation is

[2] It is possible to cast an entirely equivalent definition in terms of control dependence, using postdominance information. However the chosen explanation here seems more concise.

removed from this edge ensuring that it will not be included in any conditional probability calculations.

We reduce the program by removing statements that have a lower execution probability than the threshold $(1 - t)$.

Note that this scheme will not produce the same results as the first scheme, in general, since the execution probability values are different. This scheme may construct *larger* slices, since there may be fewer probability-annotated edges after the initial slice.

2.3.4 Concurrent Application of Dependence Information and Probability Information

The first two schemes are inefficient, in that they analyse too much of the program to begin with. An optimal scheme would consider both dependence and probability information at once. The algorithm would be more complicated, but the performance would be more efficient.

3 Applications

There are several potential applications for probabilistic program slicing.

3.1 Speculative Thread-Level Parallelism

With the advent of commodity multi-core processors, there is an abundance of cheap parallelism to be exploited. Existing sequential programs often cannot take advantage of these parallel resources, so it is necessary to employ techniques such as *speculative thread-level parallelism* (STLP). For instance, in the context of Java programs executing on a virtual machine platform, it should be possible to predict the outcomes of methods without executing them. A new (non-speculative) thread can be forked to execute the method in the background, and the existing thread can speculatively execute the code that follows the method return point, assuming that it has correctly predicted the method's behaviour. The simplest scheme that performs this method-level speculation relies on value prediction techniques [LS96] to predict the method's return value [CO98, HBJ03, PV04].

However, return value prediction fails to capture other side-effects that a method may cause. These often cause dependence violations that require the speculation to be aborted, and non-speculative execution to be restarted. A more sophisticated method effect prediction technique could involve probabilistic program slicing to construct a 'reduced' version of the method to be executed speculatively. Figure 5 represents these concepts diagrammatically.

The Mitosis system [QMS+05] already implements a speculation scheme similar to the one outlined in this section, with so-called 'speculative p-slices' being constructed and executed at the start of speculative threads.

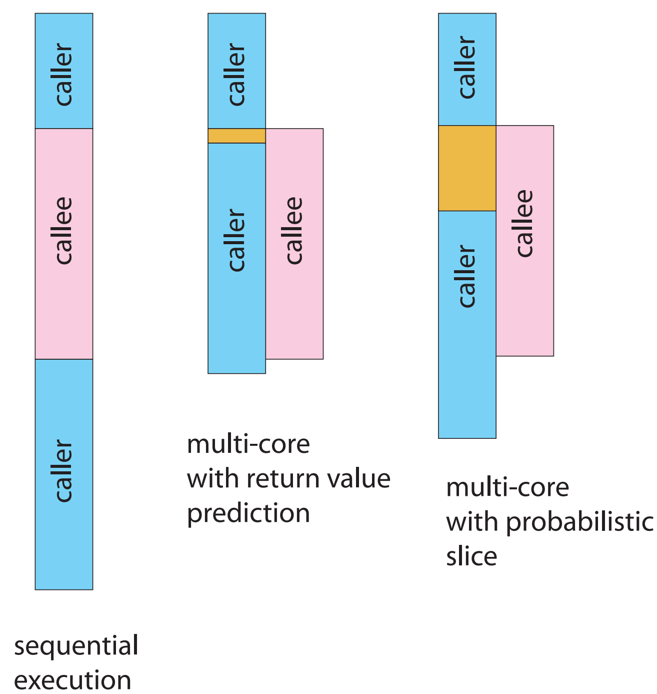

Figure 5: Sequential and speculative versions of method call sequence, time increases vertically downwards. In the sequential case, the callee is executed completely before the caller can continue. In the multi-core cases, the method's effect is somehow speculatively predicted and the caller continues on one core while the callee is executed on another core to validate the speculation. A more accurate method effect prediction such as a probabilistic slice may take longer to execute, but it means that mis-speculation is less likely to occur.

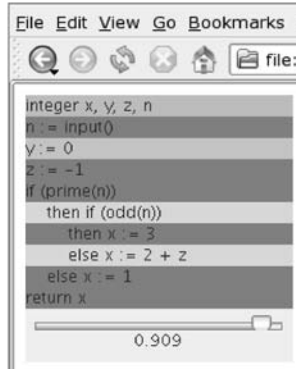

Figure 6: Screenshot of simple program comprehension applet, different colours highlight status of different statements in the program at the specified probability threshold level

3.2 Program Comprehension

It is often important for developers to discover how control flows through their program in common cases. Probabilistic program slicing could be integrated into a program comprehension framework to achieve this. For example, the developer may be able to specify a slicing criterion, then have a slider bar to specify the probability that the slice is correct. More of the program could be highlighted as the slider bar moves to increase the probability, indicating that the slice grows as the probability of correctness increases. Figure 6 shows a simple prototype GUI, operating on the program from Figure 1. This technology would rely on simple data tables such as that in Figure 3, together with calculations as outlined in Section 2.3.2.

3.3 Test Case Generation

The information computed for probabilistic program slicing can also be useful for path selection for software test data generation. In some cases it will be best to test the most frequently executed paths, whereas in other cases it may be desirable to test the least frequently executed paths. This could be easily handled by reversing the probabilities of each conditional branch outcome.

4 Related Work

Maruyama and Shima describe a program analysis in which the amount of dependence is quantified. They represent this information using *weighted program dependence graphs* [MS99]. However, they are measuring the changes in dependence information over the development history of the source code, rather than over the execution history of the running program. They use their information

to extract reduced methods that can be used for refactoring transformations. It seems possible that their framework could be adapted to express dependence probabilities, and this formalism could be used for probabilistic program slicing.

Conditioned slicing is a well-established discipline [CCL98, FDHH04]. The slicing criterion constrains program inputs in such a way as to set some conditional branch probabilities to 0. As such, conditioned slicing may be seen as a form of probabilistic slicing, however it only deletes branches whose outcomes are impossible given the input constraints, whereas we also delete branches whose outcomes are unlikely yet still possible.

As already mentioned, the *Mitosis* system [QMS+05] already implements some form of probabilistic slicing for extremely effective STLP optimization. They do not give full details of their algorithm however. Also they do not show how probabilistic slicing ideas may be applied to other areas.

5 Future Work

Much work remains to be done if probabilistic program slicing is to become a mainstream technique for the slicing community.

1. The algorithm has to be formalized properly.
2. The algorithm has to be adapted (scaled up) to handle constructs from more realistic programming languages.
3. Section 2.3 indicates that there is an clearly an issue about phase-ordering. In conditioned slicing, the program is generally ‘conditioned’ before it is sliced. This ‘conditioning’ phase is similar to partial evaluation [JGS93]. We are not sure whether the the probabilistic slicing algorithm should similarly be separated into two distinct phases.
4. A proof-of-concept implementation is required, for each of the applications outlined in Section 3. Since the author has considerable experience with the Jikes RVM [AAB+00, AAB+05] system for Java programs, this seems to be a good starting place.

References

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb 2000.

[AAB+05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp.

The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, Feb 2005.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595–607, Dec 1998.

[CHH+03] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, 2003.

[CHM05] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112:149–166, 2005.

[CO98] M.K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, 1998.

[Dan99] Sebastian Danicic. *Dataflow Minimal Slicing*. PhD thesis, University of North London, 1999.

[FDHH04] Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software: Practice and Experience*, 34(1):15–46, Jan 2004.

[HBJ03] Shiwen Hu, Ravi Bhargava, and Lizy Kurian John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5, Nov 2003.

[HP03] John L. Henessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[KNP04] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the First International Conference on Quantitative Evaluation of Systems*, pages 322–323, 2004.

[KNP05] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic model checking in practice: Case studies with PRISM. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):16–21, Mar 2005.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, 1996.

[MS99] Katsuhisa Maruyama and Kenichi Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 236–245, 1999.

[ÖNG04] Emre Özer, Andy Nisbet, and David Gregg. Stochastic bit-width approximation using extreme value theory for customizable processors. In *13th International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 250–264, 2004.

[OO84] K. Ottenstein and L. Ottenstein. The program dependence graph in software development environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[PHW05] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic λ-calculus and quantitative program analysis. *Journal of Logic and Computation*, 15(2):159–179, 2005.

[PV04] Christopher J.F. Pickett and Clark Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 40–47, 2004.

[PW00] Alessandra Di Pierro and Herbert Wiklicky. Concurrent constraint programming: towards probabilistic abstract interpretation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 127–138, 2000.

[QMS+05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 269–279, 2005.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.