Context-Sensitivity Matters, But Context Does Not

Jens Krinke
FernUniversität in Hagen
Hagen, Germany

Abstract

Whether context-sensitive program analysis is more effective than context-insensitive analysis is an ongoing discussion. There is evidence that context-sensitivity matters in complex analyses like pointer analysis or program slicing. One might think that the context itself matters, because empirical data shows that context-sensitive program slicing is more precise and under some circumstances even faster than context-insensitive program slicing. Based on some experiments, we will show that this is not the case.

The experiment requires backward slices to return to call sites specified by an abstract call stack. Such call stacks can be seen as a poor man's dynamic slicing: For a concrete execution, the call stack is captured, and static slices are restricted to the captured stack. The experiment shows that there is no significant increase in precision of the restricted form of slicing compared to the unrestricted traditional slicing. The reason is that a large part of an average slice is due to called procedures.

1. Introduction

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 [20, 21]. The other main approach to slicing uses reachability analysis in program dependence graphs (PDGs) [7]. 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. through 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 in three simple steps: Map the slicing criterion on a node, find all backward reachable nodes, and map the reached nodes back on the statements.

The extension of the PDG for *interprocedural programs* introduces more nodes and edges: For every procedure a *procedure dependence graph* is constructed, which is basically a PDG with *formal-in* and *-out* nodes for every formal parameter of the procedure. A procedure call is represented by a *call* node and *actual-in* and *-out* nodes for each actual parameter. The call node is connected to the entry node by a *call* edge, the *actual-in* nodes are connected to their matching *formal-in* nodes via *parameter-in* edges, and the *actual-out* nodes are connected to their matching *formal-out* nodes via *parameter-out* edges. Such a graph is called *Interprocedural Program Dependence Graph (IPDG)*. The *System Dependence Graph (SDG)* is an IPDG, where *summary edges* between actual-in and actual-out have been added representing transitive dependence due to calls [10].

To slice programs with procedures, it is not enough to perform a reachability analysis on IPDGs or SDGs. The resulting slices are not accurate as the *calling context* is not preserved: The algorithm may traverse a parameter-in edge coming from a call site into a procedure, traverse some edges there, and finally traverse a parameter-out edge going to a different call site. The sequence of traversed edges (the path) is an *unrealizable path*: It is impossible for an execution that a called procedure does not return to its call site. We consider an interprocedural slice to be *precise* if all nodes included in the slice are reachable from the criterion by a *realizable* path. Precise interprocedural slices can efficiently be computed by a two-pass algorithm that relies on summary edges [10].

The next section will discuss empirical results and related work on how context-sensitive program slicing compares to context-insensitive slicing. Section 3 contains a new form of program slicing that restricts the slice to obey a specified calling context. That approach is used for an experiment in Section 4 to argue about context-sensitivity and context. The counter-intuitive results of the experiment are explained before the last section draws conclusions.

2. Previous Results

There has been some debate whether the increased precision is worth the increased complexity of context-sensitive program analysis. There is no final conclusion as every program analysis differs. For pointer analysis, context-sensitive and context-insensitive analyses exist, however, many authors claim that context-sensitive pointer analysis is too expensive for only a small increase in precision [9, 8].

The case is different for program slicing: For slicers that use a Weiser-style algorithm based on data flow equations, context-sensitive slicing is expensive. The experiments presented in [4, 16] show that unlimited context-sensitive Weiser-style slicing is not affordable; Mock et al [16] limit the depth of the considered context to two. This means that the slicing algorithm only returns for a chain of two call sites to the correct call site and is context-insensitive after that. With this limited context-sensitivity the conducted experiments show no large increase in precision. These results are in contrast to at least three experiments done with PDG-based slicing. The first, done by Agrawal and Guo [1], presented results stating that context-sensitive slicing is faster and more precise than context-insensitive slicing. However, their results are doubtful because this approach has been shown to be incorrect in the second large study performed by Krinke [12]. There, the experiments showed that context-sensitive slicing in the style of Horwitz et al [10, 19] is always much more precise than context-insensitive slicing. On average, the slices computed by the context-insensitive slicing algorithm are 67% larger than the ones computed by the context-sensitive algorithm. Moreover, the context-insensitive algorithm is even slower; on average, it needs 23% more time. The third large scale study performed by Binkley and Harman [6] had similar results based on a very large set of test cases. Their algorithm is also based on PDGs and uses the original two-pass algorithm [10] implemented in the CodeSurfer slicing tool [3]. Their results showed that context-insensitive slices are on average 50% larger than their context-sensitive counterparts. The results are not directly comparable to Krinke's results, because of different ways to measure slice sizes.

The results for PDG-based slicing contradict the ones presented by Mock et al [16]. Therefore, Krinke [12] also experimented with PDG-based slicing algorithms that rely on explicit context-sensitivity and handle it similar to Weiser-style algorithms. The performed experiments limit the depth of the context similar to the approach of Mock et al. To assess the results with limited context, Krinke compared the size of the computed slices against the ones computed by the context-insensitive and the (unlimited) context-sensitive algorithm. He considered the size of the slices computed by the context-insensitive algorithm as 0% precision and the size of the slices computed by the unlimited context-sensitive algorithm as 100% precision. For experiments done with different limits, he reported increasing numbers of precision. For example, even a limit of one results in an average precision of 63% and a limit of 6 already reaches 98% precision.

Binkley [5] defines *calling-context slices* to include those statements that influence the criterion in a specific calling context, but no other calling context. This algorithm is used in regression test selection and optimization. However, Binkley does not report empirical data on his algorithms. His algorithm is very similar to the one presented next, but imprecise as shown at the end of the next section.

3. Context-Restricted Slicing

The results of the PDG-based slicing studies suggest that context matters in slicing algorithms, and that context-sensitive algorithms have an enhanced precision with decreased computation time. This may lead to the assumption that the context itself is the reason for precision. This leads us to the creation of a "poor man's dynamic slicer". During debugging, the programmer is not interested in all possible executions, but in one specific, e.g. if we want to find out why a program crashed at a certain point. Because static slicing does not consider a specific execution but all possible executions, it does not suit such debugging tasks very well. Instead, dynamic slicing [11] has been developed; it computes slices which are specific to one particular execution. Because of this restriction, dynamic slices are more precise than static slices. However, the computation of dynamic slices is expensive and has to be redone for every performed execution.

As yet, no ready-to-use dynamic slicer is available. Instead, one has to rely on one of the available static slicers like CodeSurfer [3], Sprite [4, 16], or Unravel [15]. This results in the following scenario: If a crashed program is debugged, we can normally extract the current call stack that leads to a crash. A simple adaptation of the slicing algorithm could force the computed slice to obey the extracted call stack by requiring called procedures to return to the calling procedure as found in the call stack.

A program analysis is context-sensitive, if it only considers interprocedurally realizable paths. One way to describe those paths is via context-free language reachability as done by Reps [18]: The intraprocedural program dependence graph can be seen as a finite automaton and the intraprocedurally realizable paths are words of its accepted language. Therefore, reachability in the program dependence graph is an instance of regular language reachability. The problem in interprocedural reachability is the proper matching of call edges to return edges. This can be achieved by defining a context-free language on top of the IPDG. First, we assume that call and actual parameter nodes are marked with a label for their call site c. Edges in the IPDG are now marked according to their source and target nodes:

- Call edges between a call node m at call site c and a node n in procedure p are marked with "$(_c$".
- Parameter-in edges between an actual-in parameter node m at call site c and a formal-in node n in procedure p are also marked with "$(_c$".
- Parameter-out edges between a formal-out node in procedure p and an actual-out node n at call site c are marked with "$)_c$".
- All other edges are marked with ϵ.

Let Σ be the set of all edge labels in an IPDG G. Every path in G induces a word over Σ by concatenating all the labels of the edges that are on the path. A path is an interprocedurally *matched* path if it is a word of the context-free language defined by:

$$\begin{array}{rcll} M & \rightarrow & MM & \\ & \mid & (_c M)_c & \forall (_c \in \Sigma \\ & \mid & \epsilon & \end{array}$$

This grammar assures the proper matching of calls and returns by simulating an abstract call stack.

Interprocedurally matched paths require their start and end node to be in the same procedure. Interprocedurally realizable paths with start and end node in different procedures have only partially matching calls and returns: Dependent on whether the end node is lower or higher in the abstract call stack, the paths are right-balanced or left-balanced. A path is an interprocedurally *right-balanced* path if it is a word of the context-free language defined by:

$$\begin{array}{rcll} R & \rightarrow & RR & \\ & \mid & M & \\ & \mid & (_c & \forall (_c \in \Sigma \\ & \mid & \epsilon & \end{array}$$

Here, every $)_c$ is properly matched to a $(_c$ to the left, but the converse need not hold. A path is an interprocedurally *left-balanced* path if it is a word of the context-free language defined by:

$$\begin{array}{rcll} L & \rightarrow & LL & \\ & \mid & M & \\ & \mid &)_c & \forall (_c \in \Sigma \\ & \mid & \epsilon & \end{array}$$

An *interprocedurally realizable path* starts as a left-balanced path, and ends as a right-balanced path:

$$I \rightarrow LR$$

Interprocedural reachability between nodes in PDGs and context-sensitive slices are now defined based on interprocedurally realizable paths:

Definition 1 (Interprocedural Reachability)
A node n is *interprocedurally reachable* from node m, iff an interprocedurally realizable path from m to n in the IPDG exists, denoted as $m \rightarrow_{\mathsf{R}}^{\star} n$.

As noted earlier, we consider an interprocedural slice to be *precise* if all nodes included in the slice are reachable from the criterion by an interprocedurally realizable path:

Definition 2 (Slice in an IPDG)
The (*backward*) *slice* $S(n)$ of an IPDG $G = (N, E)$ at node $n \in N$ consists of all nodes on which n (transitively) depends via an interprocedurally realizable path:

$$S(n) = \{m \in N \mid m \rightarrow_{\mathsf{R}}^{\star} n\}$$

These definitions cannot be used in an algorithm directly, because it is impractical to check if paths are interprocedurally realizable.

Accurate slices can be calculated with a modified algorithm on SDGs [10]: The benefit of SDGs is the presence of *summary* edges that represent transitive dependence due to calls. Summary edges can be used to identify actual-out nodes that are reachable from actual-in nodes by an interprocedurally realizable path through the called procedure without analyzing it. The idea of the slicing algorithm using summary edges [10, 19] is to first slice from the criterion *ascending* into calling procedures, and then to slice from all visited nodes *descending* into called procedures.

Now, we restrict an interprocedurally realizable path to a call stack s. A call stack s is represented by a list of call sites c_i: $s = \langle c_1, \ldots, c_k \rangle$. A path *matches a call stack* s if it is a word of the context-free language induced by $s = \langle c_1, \ldots, c_k \rangle$:

$$\begin{array}{rcl} I & \rightarrow & L \\ & \mid & L(_{c_k} M \\ & \mid & L(_{c_{k-1}} M (_{c_k} M \\ & \vdots & \\ & \mid & L(_{c_1} M \ldots (_{c_k} M \end{array}$$

This requires the path to return to the chain of call sites in the call stack if there is no matching call.

Definition 3 (Context-Restricted Slice)
The (*backward*) *slice* $S(n, s)$ of an IPDG $G = (N, E)$ at node $n \in N$ restricted to the call stack s consists of all nodes on which n (transitively) depends via an interprocedurally realizable path that matches the call stack s:

$$S(n, s) = \{m \in N \mid m \overset{s}{\rightarrow}_{\mathsf{R}}^{\star} n\}$$

Here, $m \overset{s}{\rightarrow}_{\mathsf{R}}^{\star} n$ denotes that there exists an interprocedurally realizable path from m to n matching s. Note that a context-restricted slice requires the criterion n to be in a procedure called from the topmost call site c_k of $s = \langle c_1, \ldots, c_k \rangle$.

Input: $G = (N, E)$ the given SDG
$n \in N$ the given slicing criterion
$s = \langle c_i, \ldots, c_k \rangle$ the given call stack
Output: $S \subseteq N$ the slice for the criterion n

$W^{\mathrm{up}} = \{n\}$
$W^{\mathrm{down}} = \emptyset$
$S = \{n\}$

first pass, ascending slice
for $i = k \ldots 1$ **do**
  handle the calling context site by site
  $W = W^{\mathrm{up}}$
  $W^{\mathrm{up}} = \emptyset$
  while $W \neq \emptyset$ *worklist is not empty* **do**
    $W = W/\{n\}$ *remove one element from the worklist*
    foreach $m \rightarrow n \in E$ **do**
      if $m \notin S$ **then**
        if $m \rightarrow n$ is a parameter-out edge ($m \stackrel{\mathrm{po}}{\rightarrow} n$) **then**
          delay the further traversal until the second pass
          $W^{\mathrm{down}} = W^{\mathrm{down}} \cup \{m\}$
          $S = S \cup \{m\}$
        elsif $m \rightarrow n$ is a parameter-in or call edge ($m \stackrel{\mathrm{pi,cl}}{\rightarrow} n$)
        and the call site of m is c_i **then**
          traversal will continue in the next iteration
          $W^{\mathrm{up}} = W^{\mathrm{up}} \cup \{m\}$
          $S = S \cup \{m\}$
        else
          $W = W \cup \{m\}$
          $S = S \cup \{m\}$

second pass, descending slice
while $W^{\mathrm{down}} \neq \emptyset$ *worklist is not empty* **do**
  $W^{\mathrm{down}} = W^{\mathrm{down}}/\{n\}$ *remove one element from the worklist*
  foreach $m \rightharpoonup n \in E$ **do**
    if $m \notin S$ **then**
      if $m \rightharpoonup n$ is not a parameter-in or call edge ($m \stackrel{\mathrm{pi,cl}}{\rightarrow} n$) **then**
        $W^{\mathrm{down}} = W^{\mathrm{down}} \cup \{m\}$
        $S = S \cup \{m\}$
return S *the set of all visited nodes*

Figure 1. Summary Information Slicing (in SDGs)

The algorithm in Figure 1 computes a context-restricted slice. It is a variant of Krinke's context-sensitive slicing algorithm [12], which is a variant of Horwitz et al's algorithm [10, 19]. Here, the first pass that computes the slices stopping at parameter-in or call edges has been changed such that it is repeated once for every call site c_i of the specified call stack. In each iteration, every node reachable via intraprocedural edges is added to the worklist W. If a parameter-out edge is traversed, the reached node is added to the worklist W^{down}, which is processed in the second pass. If parameter-in or call edges are traversed, the reached node has to be part of the current call site c_i. If that is the case, the reached node is added to the worklist W^{up}, which is used as the initial worklist W for the next iteration that processes call site c_{i-1}. All intraprocedural edges are traversed in the current iteration.

The presented algorithm computes context-restricted slices which are defined almost identical to Binkley's calling-context slices [5]. The algorithm he presented basically computes a slice without the first pass for each call site in the calling context: Starting with the criterion, a (second pass) slice is computed ignoring edges that lead into a calling procedure, but traversing edges that lead into called procedures. The reached formal-in nodes are extracted from the computed slice, and mapped to the corresponding actual-in nodes at the next call site of the current calling context. The call site is removed from the calling context, and the set of actual-in nodes is used as the slicing criterion for the next iteration, which is repeated until the calling context is empty. This algorithm is imprecise, because it mixes the first pass with the second. Consider the following example:

```
 1  int g() {
 2    a = ...;
 3    return f(a);
 4  }
 5
 6  int f(x) {
 7    if (...)
 8      b = f(0);
 9    return x;
10  }
```

The corresponding PDG is shown in Figure 2. We compute the backward slice for variable `b` in line 8 (node 12) within the calling context $\langle \bot, 3 \rangle$ ($\bot$ represents the invocation from the runtime system). Binkley's algorithm starts by computing a slice that traverses the parameter-out edge between nodes 12 and 13 due to the recursive call of function `f` (nodes 7–13). It then extracts the formal-in node 8 for `x` in line 6, and maps it to the actual-in node 4 for `a` in line 3, because the current call site is 3. The subsequent iteration adds nodes 1–4 and thus, line 2 to the slice. However, this line should not be in the slice, because line 2 never has an influence on line 8.

The algorithm in Figure 1 starts with the first iteration for call site 3. It also encounters the parameter-out edge between nodes 12 and 13 due to the recursive call, but delays the further traversal to the second pass. It encounters the call edge between nodes 3 and 7 and delays the further traversal to the second iteration. Thus, the first iteration inserts nodes 7 and 9–12 into the slice. The next iteration (call site $\bot$) continues at node 3 and adds nodes 1 and 3 to the slice. As no interprocedural edges are encountered and the calling context has been completely traversed, the first pass is complete. The second pass starts at node 13 and adds nodes 8 and 13 to the slice. The encountered interprocedural edges are ignored and thus, the second pass is complete. The final slice consists of nodes 1, 3, and 7–13. It does not include line 2 (node 2).

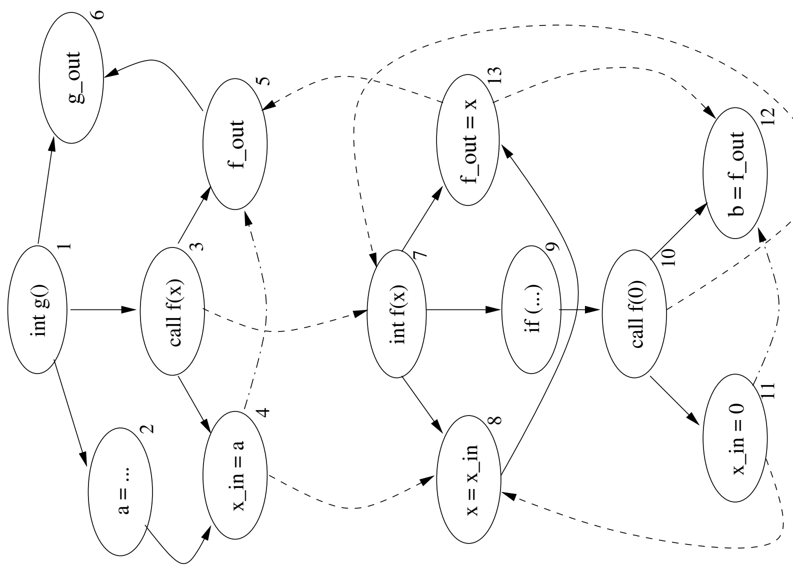

Figure 2. A PDG example

4. Experiment

We have implemented the above algorithm in our slicing infrastructure [14, 12, 13] and performed two case studies, based on the programs `ctags` and `patch`. For each of the programs we performed one characteristic execution in a debugger. On every execution of a procedure, we dumped and extracted the current call stack. This produced one set of call stacks for each program. We then computed backward slices for each formal-in parameter node of the intercepted procedure for each call stack. We computed each slice twice, once using context-restricted slicing and once using traditional context-sensitive slicing. The results are shown in Table 1 for the two test cases `ctags` (left column) and `path` (right column). The first two rows show the number of unique call stacks extracted from the test execution, and the number of computed slices. The remaining rows show the average slice size measured in SDG nodes and as a percentage of the SDG.

The results do not confirm the expected effect: context-restricted slices do not have a much higher precision than context-sensitive slices. Though the context-restricted slice is 9% smaller for `ctags` than the context-sensitive slice, the percentage of the complete program just decreases from 20% to 19%. For the other test `patch`, the average size for the context-restricted slice is just 1% smaller, and the difference in percentage of the complete program is negligible.

So the question is why there is just a small size reduction? Our hypothesis is that this is related to unrestricted called procedures. Context-restricted slices only restrict the

	ctags	patch
unique stacks	186	85
slices	4136	2569
average size context-sensitive slice	2100	7109
average size context-restricted slice	1914	7021
average size context-sensitive slice	20%	34%
average size context-restricted slice	19%	34%
average size reduction	9%	1%

Table 1. Average sizes of context-sensitive and -restricted slices

calling-context of calling procedures. The context of called procedures is not restricted (as long as called procedures are handled context-sensitively). We believe that a large part of an average slice is due to called procedures. To investigate this, we repeated the experiment with *truncated* backward slicing. A truncated (backward) slice does not contain nodes from called procedures; it does not descend into them. To compute it, the second pass of the slicing algorithm is left out (because it computes exactly those nodes). A truncated (backward) slice is computed by always ignoring the parameter-out edges, as this would process called procedures during backward traversal. The algorithm in Figure 1 can be adapted to the truncated version accordingly: We remove the second pass and the first branch of the if-elsif-then-cascade. With this modification, we repeated the experiment; Table 2 shows the result. We can see that the average truncated slice is much smaller than a non-truncated slice. For `ctags`, the size went down from 2100 to 924 nodes of the SDG (a 56% reduction), and for `patch`, it went down from 7109 to 2916 nodes (a 59% reduction). This illustrates that the majority of nodes in a slice are nodes of called procedures. The numbers for context-restricted slices now support our hypothesis: For `ctags`, the context-restricted slice is on average half the size of a context-sensitive slice, and for `patch`, it is still 42% smaller. However, we still expected a larger reduction because of the following observation: For `ctags`, we determined that on average, every procedure is called from three different call sites, and for `patch`, we measured 4.8 different call sites. We also measured the average size of the call stacks, which is 8.5 for `ctags` and 4.2 for `patch`. These numbers suggest that a context-sensitive truncated slice would visit many more procedures than a context-restricted one. However, values around 50% suggest that there are not so many alternative call stacks that can lead to a specific point of execution. A further investigation of this topic is planned.

It must be noted that both programs had no function pointers. It is expected that for programs making heavy use of function pointers the results may be different, because

	ctags	patch
average size context-sensitive slice	924	2916
average size context-restricted slice	463	1701
average size context-sensitive slice	9%	14%
average size context-restricted slice	4%	8%
average size reduction	50%	42%

Table 2. Average sizes of for truncated slices

function pointers have a strong influence on the precision of static slices. The same is expected for object oriented programs due to dynamic binding.

5. Conclusions

The presented approach of context-restricted slices can efficiently be implemented in current static slicing tools that are based on PDGs. For debugging, context-restricted slicing can be used as a poor man's dynamic slicer. However, the size reduction is not large enough for the non-truncated slices. We plan to integrate and experiment with other lightweight approaches like approximate dynamic slicing [2] that captures whether a statement corresponding to a node in the PDG has ever been executed, or call-mark slicing [17], where it is captured whether a procedure has ever been executed.

The presented experiment adds another aspect to the discussion about context-sensitive or context-insensitive program analysis. For program slicing, earlier studies showed evidence that context-sensitive slicing algorithms are much more precise and can even be faster than their context-insensitive counterparts. However, the experiment of this approach shows that restricting slices to specific contexts does not lead to significant smaller slices.

We do not claim that this is a general result, as the experiment is to small for that purpose. To draw a generally valid conclusion, this experiment has to be repeated in a larger scale, like done by Binkley and Harman [6] or Krinke [12, 13]. Such an experiment has to gather more statistical data, e.g. variance of the different slice sizes. Additionally, the results are only valid for C—context plays a different role in object-oriented programming languages, and we expect different results.

Acknowledgments. David Binkley provided valuable comments.

References

[1] G. Agrawal and L. Guo. Evaluating explicitly context-sensitive program slicing. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12, 2001.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on*

Programming Language Design and Implementation, pages 246–256, 1990.

[3] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.

[4] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[5] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, Aug. 1997.

[6] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *International Conference on Software Maintenance*, pages 44–53, 2003.

[7] 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.

[8] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.

[9] M. Hind and A. Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[10] 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.

[11] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[12] J. Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, pages 22–31, 2002.

[13] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, Apr. 2003.

[14] 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.

[15] J. Lyle and D. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In *Proceedings of Software Quality Week*, 1997.

[16] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, 2002.

[17] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: An efficient and economical way of reducing slice. In *International Conference of Software Engineering*, pages 422–431, 1999.

[18] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

[19] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

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

[21] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.