Effects of Context on Program Slicing

Jens Krinke

FernUniversität in Hagen, Fachbereich für Elektrotechnik und Informationstechnik, 58084 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. Empirical data shows that context-sensitive program slicing is more precise and under some circumstances even faster than context-insensitive program slicing. This article will add to the discussion by examining if the context itself matters, i.e. if a given context leads to more precise slices for that context. Based on some experiments, we will show that this is strongly dependent on the structure of the programs.

The presented experiments require 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 experiments show that for recursive programs there is a large increase in precision of the restricted form of slicing compared to the unrestricted traditional slicing.

The same experiments also show that a large part (more than half) of an average slice is due to called procedures.

Key words: Program Analysis, Program Slicing

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 [1,2]. The other main approach to slicing uses reachability analysis in program dependence graphs (PDGs) [3]. Program dependence graphs mainly consist of nodes representing the statements of a program as well as control and data dependence edges:

† This article appears in the Journal of Systems and Software. The original publication is available at `http://dx.doi.org/10.1016/j.jss.2006.02.040`

- 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 [4].

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 [4].

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 follow a specified calling context. That approach is used for some experiments in Section 4 to argue about context-sensitivity and context. The results of the experiments 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 [5,6].

The case is different for program slicing: Three large experiments with PDG-based program slicing have shown that context-sensitive slicing is much more precise and can even be faster than context-insensitive slicing. The first, done by Agrawal and Guo [7], 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 (slices can be too small) in the second large study performed by Krinke [8]. Krinke's experiments showed that context-sensitive slicing in the style of Horwitz et al [4,9] 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 [10] 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 [4] implemented in the CodeSurfer slicing tool [11]. 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, however, the results are similar.

The results for PDG-based slicing contradict the ones for Weiser-style slicing. For slicers that use a Weiser-style algorithm based on data flow equations, context-sensitive slicing is expensive. The experiments presented by Atkinson et al [12] and Mock et al [13] show that unlimited context-sensitive Weiser-style slicing is not affordable; Mock et al [13] 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 the three experiments above done with PDG-based slicing. Therefore, Krinke [8] 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.: The analysis is context-sensitive for the last k stack frames and context-insensitive for the rest. 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 $k = 1$ (the analysis is context-sensitive for the current stack frame) results in an average precision of 63% and a limit $k = 6$ already reaches 98% precision. However, Krinke's results in this experiment sup-

ports Mock's and Atkinson's results that slicing algorithms which handle calling context explicitly are too expensive for large contexts, because every procedure has to be analyzed once for each calling context of the procedure. In PDG-based slicing that uses algorithms similar to Horwitz et al's [4], each procedure has to be analyzed only once or twice.

Binkley [14] 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 [15] 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, because the necessary information has to be computed for every execution of a statement, i.e. the time needed to compute dynamic slices is dependent one the length of the execution and not (only) on the size of the program as in static slicing.

As yet, no ready-to-use dynamic slicer is available. Instead, one has to rely on one of the available static slicers like CodeSurfer [11], Sprite [12,13], or Unravel [16]. 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 follow the extracted call stack by requiring called procedures to return to the calling procedure as found in the call stack. To do that, we will first revisit the formal definitions of interprocedurally realizable paths and later adapt the definitions to context-restricted interprocedurally realizable paths.

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 [17]: 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:

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

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:

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

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:

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

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 may reach 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 [4]: 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 [4,9] is to first slice from the criterion *ascending* into calling procedures (i.e. traverse paths along R), and then to slice from all visited nodes *descending* (i.e. along L) 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 $s = \langle c_1, \ldots, c_k \rangle$:

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

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*:*

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

Here, $m \overset{s}{\to}^{\star}_{\mathsf{R}} n$ *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* $s = \langle c_1, \ldots, c_k \rangle$*.*

The algorithm in Figure 1 computes a context-restricted slice. It is a variant of Krinke's context-sensitive slicing algorithm [8], which is a variant of Horwitz et al's algorithm [4,9] and utilizes summary edges. 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 work-list W. If a parameter-out edge is traversed, the reached node is added to the work-list 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 work-list W^{up}, which is used as the initial work-list W for the next iteration that processes call site c_{i-1}. All intraprocedural edges are traversed in the current iteration. Both passes can be matched to the definition of the context-free language for restricted interprocedurally realizable paths: The first pass basically traverses the paths along the $(_{c_k} M$, $(_{c_{k-1}} M(_{c_k} M$, etc. but skips the M along summary edges. The second pass traverses the paths along L and M like Horwitz et al's algorithm [4,9].

The presented algorithm in Fig. 1 computes context-restricted slices which are defined almost identical to Binkley's calling-context slices [14]. The algorithm he presented basically computes a slice without the ascending first pass for each call

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

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

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

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

Fig. 1. Context-Restricted Slicing (in SDGs)

site in the calling context: Starting with the criterion, a (descending 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 (this is the ascending step). 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 descending and the ascending steps. Consider the following example:

```
 1  int main() {
 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 at node 12 within the context $\langle \bot, 3 \rangle$. 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).

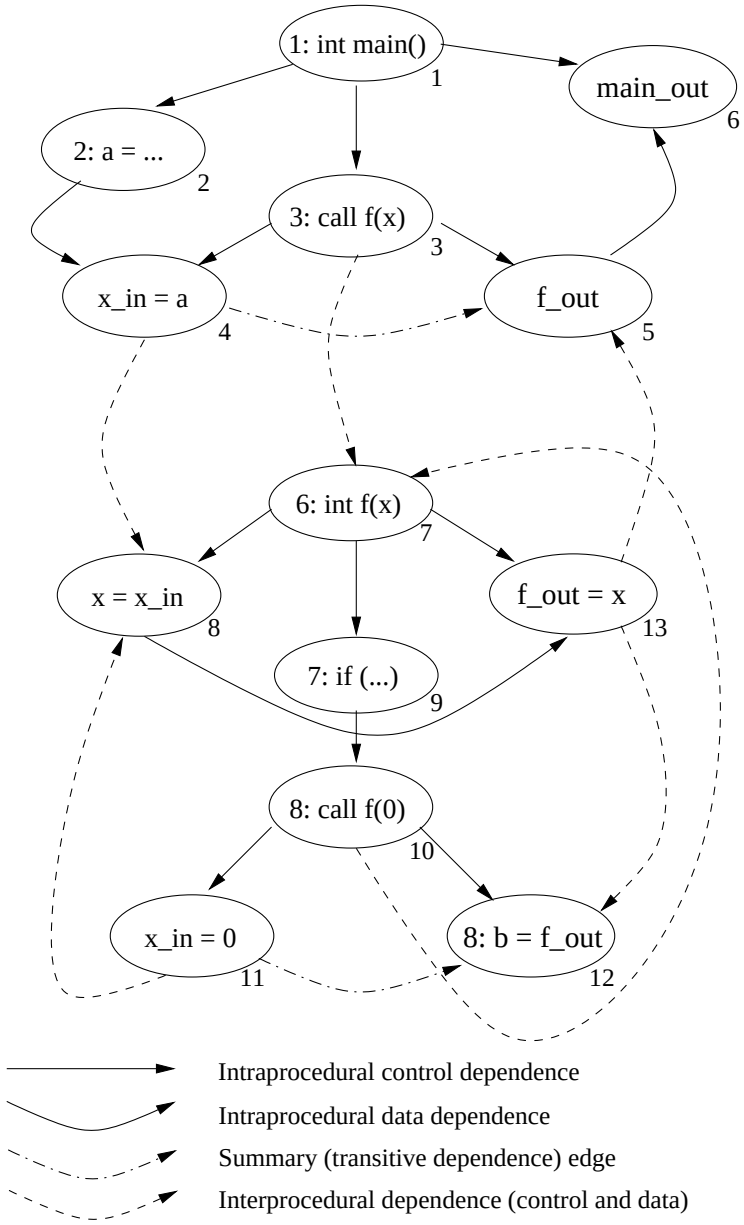

Fig. 2. A PDG example

4 Evaluation

We have implemented the above algorithm (Fig. 1) in our slicing infrastructure [18,8,19] and performed an initial experiment to validate the presented approach for context-restricted slicing [20]. This initial experiment is presented next. A larger evaluation follows that contrasts some of the results of the initial experiment.

	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

4.1 Initial Experiment

The initial experiment 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 $s = \langle c_1, \ldots, c_k \rangle$, where c_1 is the call of `main` from the runtime system and c_k is the call site that called the currently executing procedure. This produced one set of call stacks for each program's execution. 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 `patch` (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 in this scenario. 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 calling-context of calling procedures in the call stack, i.e. active procedures. The context of called procedures is not restricted (as long as called procedures are handled context-sensitively), because such calls happened before the

[*] We used formal-in nodes to omit effects of the current procedure and to maximize the differences between restricted and non-restricted slices.

	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

current call stack and the called procedures are no longer active. 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 (another experiment in Section 4.2.5 will support this observation). 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. To examine this effect in detail, we performed a larger evaluation which is presented next.

	LOC	proc.	nodes	edges	summary	%
agrep	3968	90	11922	35713	12343	26
ansitape	1744	76	6733	18083	12746	41
assembler	3178	685	13393	97908	114629	54
bison	8313	161	25485	84794	29739	26
cdecl	3879	53	5992	17322	9089	34
compiler	2402	49	15195	45631	58240	56
ctags	2933	101	10042	24854	20483	45
diff	13188	181	46990	471395	612484	57
flex	7640	121	38508	235687	144496	38
football	2261	73	8850	30474	17605	37
gnugo	3305	38	3875	10657	2064	16
patch	7998	166	20484	104266	83597	44
rolo	5717	170	37839	264922	170108	39
simulator	4476	283	9143	22138	5022	18
average	5072	161	18175	104560	92332	38

Table 3
Details of the test programs

4.2 Second Experiment

After the initial experiment we performed a larger evaluation based on the programs we have already used for previous evaluations [8,19]. The details of the analyzed programs are shown in Table 3. The programs stem from three different sources: `ctags`, `patch` and `diff` are the GNU programs. The rest are from the benchmark database of the PROLANGS Analysis Framework (PAF) [21]. The 'LOC' column shows lines-of-code (measured via `wc -l`), the 'proc.' column the amount of procedures (the number of entry nodes in the PDG) and the 'nodes' and 'edges' columns show the number of nodes and edges in the IPDG. The 'summary' column contains the number of summary edges in the corresponding SDG to give an impression of the effect of calling context. The last column gives the percentage of the edges which are summary edges.

The goal was to check the validity of the results of the initial experiment for non-truncated slicing. Because a larger scale evaluation was not possible with the approach of the initial experiment, this evaluation has been done differently. Instead of extracting call stacks from executions, we generated possible call stacks from the static call graph of the programs. For recursive programs this causes a problem,

depth	1	2	3	4	5	6	7	8	9	10	11	12
`assembler`												
stacks	1	4	48	115	270	408	525	566	566	566	566	566
slices	153	439	1078	2661	3996	5509	5964	6149	6149	6149	6149	6149
avg. rest.	11	1066	3329	4222	4508	4714	4808	4868	4868	4868	4868	4868
avg. size	11	1066	3828	4507	5128	5488	5589	5628	5628	5628	5628	5628
reduction	0%	0%	13%	6%	12%	14%	14%	14%	14%	14%	14%	14%
`cdecl`												
stacks	1	8	13	426	911	1129	1163	1163	1163	1163	1163	1163
slices	101	393	757	2312	3969	4761	4895	4895	4895	4895	4895	4895
avg. rest.	824	826	843	817	844	867	877	877	877	877	877	877
avg. size	824	842	869	1156	1289	1363	1378	1378	1378	1378	1378	1378
reduction	0%	2%	3%	29%	35%	36%	36%	36%	36%	36%	36%	36%
`patch`												
stacks	1	64	375	684	1096	1420	1775	2057	2246	2385	2397	2397
slices	325	1612	5078	8291	11473	13880	15869	17321	18583	19124	19184	19184
avg. rest.	11	7010	8365	8622	8801	8894	8990	9048	9097	9113	9116	9116
avg. size	11	7196	8938	9346	9567	9687	9765	9808	9832	9844	9845	9845
reduction	0%	3%	6%	8%	8%	8%	8%	8%	8%	7%	7%	7%
`simulator`												
stacks	1	5	26	73	250	815	2310	3522	8952	8952	8952	8952
slices	64	149	382	824	1990	4158	9327	14535	28437	28437	28437	28437
avg. rest.	11	156	1334	2607	3387	3659	3758	3994	4188	4188	4188	4188
avg. size	11	156	1878	3198	3918	4288	4476	4510	4600	4600	4600	4600
reduction	0%	0%	29%	19%	14%	15%	16%	11%	9%	9%	9%	9%

Table 4
Context-restricted vs. traditional slicing (non-recursive programs)

because they have an infinite number of possible call stacks. Therefore, we limit the depth of the generated call stacks, i.e. calls are ignored after a predefined depth has been reached. Again, we selected the formal-in nodes of the procedure reached by the current call stack as slicing criteria. Tables 4, 5, 6, and 7 show the results of the evaluation for non-truncated slicing in three groups. Each program has five rows: The first row contains the number of generated call stacks for the depths 1 to 12. The second row shows the number of computed slices. The third row gives

1	2	3	4	5	6	7	8	9	10	11	12
`bison`											
1	13	96	279	681	1274	1892	2481	3070	3659	4248	4837
460	1218	2956	6200	12683	21119	28634	35421	42217	49018	55814	62615
6	468	1031	1756	2088	2132	2065	1942	1748	1530	1347	1203
6	521	1594	3027	4209	4973	5550	5993	6300	6520	6688	6819
0%	10%	35%	42%	50%	57%	63%	68%	72%	77%	80%	82%
`ctags`											
1	9	30	59	98	140	276	550	1420	3406	6430	9784
124	405	722	1048	1829	3315	6535	12807	24815	44065	69443	96853
11	513	976	1205	1728	2098	2293	2358	2374	2361	2327	2280
11	516	1005	1248	1808	2190	2479	2570	2612	2617	2629	2645
0%	1%	3%	3%	4%	4%	8%	8%	9%	10%	12%	14%
`flex`											
1	6	66	331	755	1409	2147	2934	3707	4480	5253	6026
325	1116	4137	15142	37036	69558	107889	148848	189573	230376	271182	311959
6	1746	4182	4657	4454	4207	4081	3997	3833	3572	3225	2886
6	2105	6766	7930	8919	9931	10917	11623	12164	12519	12768	12951
0%	17%	38%	41%	50%	58%	63%	66%	69%	72%	75%	78%
`rolo`											
1	43	145	322	611	1177	2148	3640	5614	8378	12257	16220
282	2325	7876	18821	34433	65986	120443	208543	329441	498120	737676	987589
6	727	3365	6103	8384	9679	10415	11110	11517	11672	11643	11421
6	8390	11214	11902	12217	12492	12719	12916	13028	13113	13190	13226
0%	91%	70%	49%	31%	23%	18%	14%	12%	11%	12%	14%

Table 5
Context-restricted vs. traditional slicing (simple recursive programs)

the average size of the context-restricted slices and the fourth row the average size of the traditional slices. The fifth row gives the size reduction in percentage. For `football` it was not possible to compute all slices above the depth 9 due to the combinatorial explosion of call stacks, thus the given numbers are approximations (we computed only a fraction of all possible slices for depths 10–12).

The set of programs are separated into two groups: the group of non-recursive pro-

1	2	3	4	5	6	7	8	9	10	11	12
`agrep`											
1	29	128	226	268	328	564	1460	5664	28472	160488	942920
203	961	2523	3619	4093	5203	8897	22829	88553	446945	2.5mio	14mio
8	1412	1922	2088	2141	2267	2002	938	328	125	72	58
8	2628	3446	3615	3691	3872	4167	4443	4592	4644	4658	4662
0%	46%	44%	42%	42%	42%	52%	79%	93%	97%	99%	99%
`ansitape`											
1	8	25	86	324	604	1461	4771	20613	98738	489363	2.4mio
87	322	879	1879	3878	5881	10201	24151	87834	400334	1.9mio	9.7mio
11	108	580	883	1068	1116	940	547	216	94	67	61
11	397	1728	2313	2506	2451	2040	1232	641	428	380	370
0%	73%	66%	62%	57%	55%	54%	56%	66%	78%	82%	84%
`compiler`											
1	3	11	42	215	545	1561	5016	18837	79417	341336	1.5mio
46	121	320	1063	4468	11362	31112	96320	347428	1.4mio	6.0mio	26mio
10	98	2114	4141	6218	6471	6659	5912	4917	4223	3917	3926
10	98	4014	6959	8084	8289	8383	8430	8459	8478	8488	8493
0%	0%	47%	41%	23%	22%	21%	30%	42%	50%	54%	54%
`diff`											
1	47	121	205	419	773	1680	3621	7285	12839	22711	38711
506	2145	4500	9412	18603	42572	83828	169857	303865	549369	947379	1.7mio
7008	7136	7233	7462	7719	8205	8373	8354	8499	7934	7494	6623
7008	7837	8587	9447	9706	9605	9699	9845	10229	10760	11402	12066
0%	9%	16%	21%	21%	15%	14%	15%	17%	26%	34%	45%

Table 6
Context-restricted vs. traditional slicing (complex recursive programs)

grams (`assembler`, `cdecl`, `patch`, and `simulator`) and the group of recursive programs. The second group is further split into two subgroups: The first group consists of programs where every recursive component contains a single recursive call (the recursive component may consist of multiple procedures, but only one procedure is called from outside the component and can be identified as the source of the recursion). The other group consists of programs with a complex recursion structure. This separation is important, as each group has its own properties that

1	2	3	4	5	6	7	8	9	10	11	12
`football`											
1	9	163	641	1197	5426	71933	1.1mio	18mio	$3 \cdot 10^8$	$5 \cdot 10^9$	$7 \cdot 10^{10}$
86	346	2140	5039	11080	85272	1.2mio	20mio	322mio	$5 \cdot 10^9$	$8 \cdot 10^{10}$	$13 \cdot 10^{11}$
11	1243	2186	2333	2563	2023	1131	786	718	*711*	*710*	*710*
11	1951	2573	2777	3514	4442	4582	4592	4592	*4592*	*4592*	*4592*
0%	36%	15%	16%	27%	55%	75%	83%	84%	*85%*	*85%*	*85%*
`gnugo`											
1	14	62	165	273	611	1887	7167	29039	119583	494031	2.0mio
30	183	499	1251	1925	4339	13929	53303	215269	881427	3.6mio	14mio
11	942	1326	1386	1328	847	446	276	223	206	199	195
11	1711	1938	2014	2030	2047	2056	2058	2059	2059	2059	2059
0%	45%	32%	31%	35%	59%	78%	87%	89%	90%	90%	90%

Table 7
Context-restricted vs. traditional slicing (complex recursive programs)—continued

influence the evaluation. For the group of non-recursive programs we can generate all possible call stacks if the depth limit is large enough. This has the effects that after a certain depth, the number of (generated) call stacks is constant (`assembler` has a maximal depth of 8, `cdecl` 7, `patch` 11, and `simulator` 9). For the group of recursive programs with a single recursive call per recursive component (called the simple recursive programs), it is not possible to generate all call stacks. However, after a certain depth has been reached, the increase of the number of call stacks is almost linear, because every added level of depth only adds a constant number of call stacks. This group consists of `bison`, `ctags`, `flex`, and `rolo`. The last group with complex recursive structures (called the complex recursive programs) is problematic for the analysis, because the increase is now polynomial. The reason is that for every added level of depth, the number of call stacks is multiplied by the number of recursive calls. This group consists of `agrep`, `ansitape`, `compiler`, `diff`, `football`, and `gnugo`.

4.2.1 Non-recursive programs

For the group of non-recursive programs, Fig. 3 shows the gain in precision for non-truncated context-restricted slicing in comparison to non-truncated unrestricted slicing dependent on the predefined depth of the call stacks. The y-axis shows the how much smaller the average restricted slice is as a percentage of the average unrestricted slice. The x-axis is for the predefined depth, ranging from a call stack of depth one to depth 12. For the depth one, there is never a difference because the

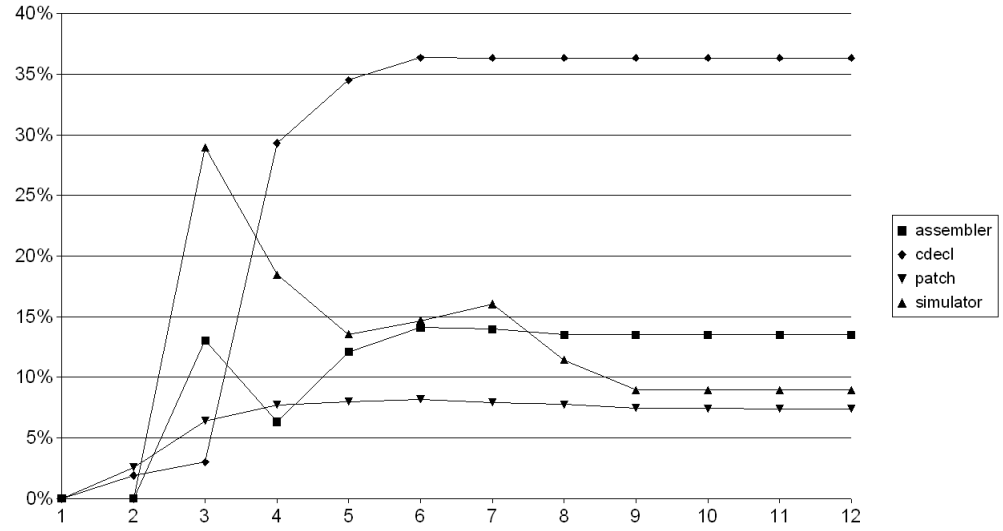

Fig. 3. Size reduction for non-recursive programs (in %)

call stack consists of the single call from the runtime system to the main procedure. All four programs have an increasing size reduction, which is constant after the predefined depth exceeds the largest possible call stack depth in the programs (`assembler` has a maximal depth of 8, `cdecl` 7, `patch` 11, and `simulator` 9). Three of the programs have only a small gain (`assembler`, `patch`, and `simulator`) between 7% and 14%. The fourth program (`cdecl`) has a larger gain of 36%. It is interesting to see that for `patch`, which has been used in the initial experiment, the gain is now larger than before. This shows that the program parts of `patch` that have not been executed during the initial part of the program are much more sensible to context-restricted slicing than the executed parts.

4.2.2 Simple recursive programs

Fig. 4 shows the gain in precision for the simple recursive programs. It is interesting to see that two programs almost have identical gains in size reduction: `flex` and `bison`. Even more interesting to note is that both programs are similar applications, one is a scanner generator and the other is a parser generator. Both programs have large gains in size reduction for larger depths. Because the recursive components dominate the numbers (as the number of non-recursive call-stacks is small in comparison to the recursive call stacks), both programs contain recursive components where the average restricted slice is much smaller than the average unrestricted slice. This is different for the other two programs. The program `ctags` has only a slow increase in the size reduction. For the call stack depth 12, the average size reduction is 14%. Note that this is again larger than in the initial experiment. Most notable is the last simple recursive program, `rolo`. It has a steep decrease in size reduction for larger call stack depths and is similar to the size reduction of `ctags` for large depths. This indicates that there is only a small difference in size between re-

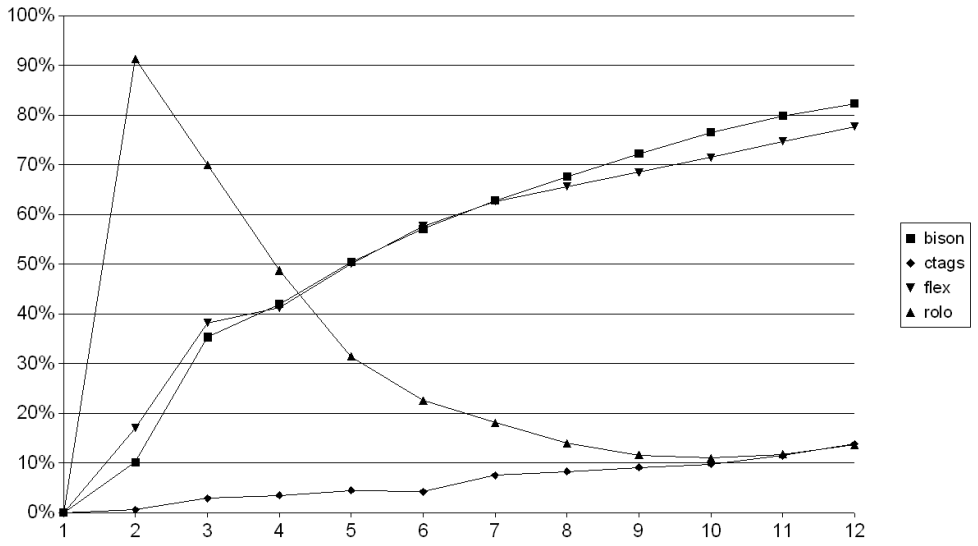

Fig. 4. Size reduction for simple recursive programs (in %)

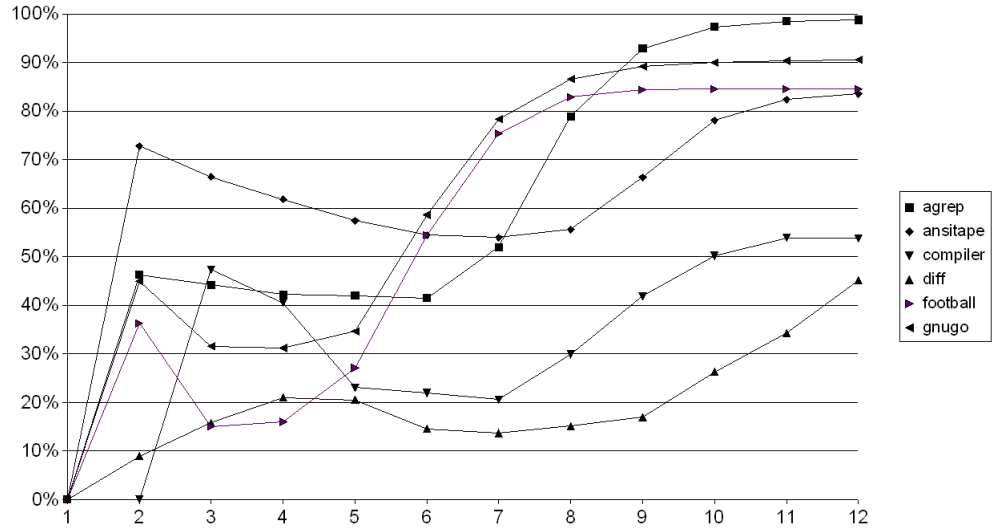

Fig. 5. Size reduction for complex recursive programs (in %)

stricted and unrestricted slices in `ctags`' and `rolo`'s recursive components. Again because the effect of the recursive components dominate.

4.2.3 Complex recursive programs

Fig. 5 shows the gain in precision for the last group of complex recursive programs. Because of the polynomial growths in the number of generated call stacks, it was only possible to analyze the programs up to a certain predefined depth. The analyses would take to much time for even larger depths. Otherwise, the results are quite similar to the results of the simple recursive programs. Most of the programs have a decreasing size reduction first, before the dominating effect of the recursive struc-

tures causes the size reduction to increase. It is not possible to predict whether `diff` will have a larger or smaller size reduction for larger depths.

4.2.4 Interpretation of the results

The presented data suggests that the previous results of the initial experiment are in contrast to the results of the larger evaluation. There are two things to consider: First, the initial experiment was based on two programs that turned out to have a small average size reduction anyway. Both programs `patch` and `ctags` have the smallest average size reduction of their group. Second, the large evaluation clearly shows a strong effect of the chosen call stack as slicing criterion on the results of restricted slicing. This suggests that call stacks and criteria in practice will have different properties than artificially generated call stacks and criteria. For example, if a program contains function pointers, many artificially generated call stacks do not occur in practice. Future research will study the effects of the differences between artificially generated and really occurring call stacks.

4.2.5 Truncated slicing

Moreover, another larger evaluation confirms the second part of the initial experiment and the drawn conclusions. Table 8 shows a comparison of the average sizes of non-truncated and truncated slices. Here, we have computed for every program all non-truncated and truncated slices (again, for each formal-in parameter as criterion). The first columns shows the name of the program, the second the number of nodes in the SDG, and the third the number of computed slices. The next two columns show the average size of a non-truncated slice as the number of nodes in the slice and the percentage of all nodes. The same is given for truncated slices in the next two columns. The last two columns presents the absolute and the relative difference. It is easy to see that the truncated slices are always much smaller than the non-truncated slices. The size reduction ranges from 50.7% (`football`) to 85.3% (`simulator`).

Because there is always a large size reduction for truncated slicing, the claim that a large share (at least half) of the average slice is due to called procedures and not calling procedures still holds.

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. In most cases, a

	nodes	slices	non-truncated		truncated		abs.	rel.
			avg. size		avg. size			
agrep	11922	1403	3183	26.7%	555	4.7%	22.0%	82.6%
ansitape	6733	1082	1645	24.4%	691	10.3%	14.2%	58.0%
assembler	13393	2401	4286	32.0%	748	5.6%	26.4%	82.5%
bison	25485	3744	1859	7.3%	332	1.3%	6.0%	82.1%
cdecl	5992	697	880	14.7%	261	4.4%	10.3%	70.3%
compiler	15195	1017	6731	44.3%	2368	15.6%	28.7%	64.8%
ctags	10042	1621	2010	20.0%	704	7.0%	13.0%	65.0%
diff	46990	10130	9179	19.5%	2308	4.9%	14.6%	74.9%
flex	38508	5191	6172	16.0%	1190	3.1%	12.9%	80.7%
football	8850	818	2593	29.3%	1278	14.4%	14.9%	50.7%
gnugo	3875	281	1798	46.4%	373	9.6%	36.8%	79.3%
patch	20484	3099	7965	38.9%	1981	9.7%	29.2%	75.1%
rolo	37839	6540	7766	20.5%	2143	5.7%	14.9%	72.4%
simulator	9143	1019	3212	35.1%	472	5.2%	30.0%	85.3%
average	18175	2789	4234	26.8%	1100	7.2%	19.6%	73.1%

Table 8
Comparison of non-truncated and truncated slices

context-restricted slice is significantly smaller than a traditional slice. For truncated slices, the size reduction is even larger. We plan to integrate and experiment with other light-weight approaches like approximate dynamic slicing [22] that captures whether a statement corresponding to a node in the PDG has ever been executed, or call-mark slicing [23], where it is captured whether a procedure has ever been executed.

The presented experiments add 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. The experiment presented here shows that restricting slices to specific contexts often leads to significant smaller slices.

Additionally, the results are only valid for C—context plays a different role in object-oriented programming languages, and we expect much stronger results for such languages.

Acknowledgments. David Binkley provided valuable comments.

References

[1] M. Weiser, Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, Ph.D. thesis, University of Michigan, Ann Arbor (1979).

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

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

[4] S. B. Horwitz, T. W. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Prog. Lang. Syst. 12 (1) (1990) 26–60.

[5] M. Hind, A. Pioli, Which pointer analysis should i use?, in: International Symposium on Software Testing and Analysis, 2000, pp. 113–123.

[6] 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.

[7] G. Agrawal, L. Guo, Evaluating explicitly context-sensitive program slicing, in: Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 6–12.

[8] J. Krinke, Evaluating context-sensitive slicing and chopping, in: Proc. International Conference on Software Maintenance, 2002, pp. 22–31.

[9] T. Reps, S. Horwitz, M. Sagiv, G. Rosay, Speeding up slicing, in: Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, 1994, pp. 11–20.

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

[11] P. Anderson, T. Teitelbaum, Software inspection using codesurfer, in: Workshop on Inspection in Software Engineering (CAV 2001), 2001.

[12] D. C. Atkinson, W. G. Griswold, The design of whole-program analysis tools, in: Proceedings of the 18th International Conference on Software Engineering, 1996, pp. 16–27.

[13] M. Mock, D. C. Atkinson, C. Chambers, 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.

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

[15] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155–163.

[16] J. Lyle, D. Wallace, Using the unravel program slicing tool to evaluate high integrity software, in: Proceedings of Software Quality Week, 1997.

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

[18] J. Krinke, G. Snelting, Validation of measurement software as an application of slicing and constraint solving, Information and Software Technology 40 (11-12) (1998) 661–675.

[19] J. Krinke, Advanced slicing of sequential and concurrent programs, Ph.D. thesis, Universität Passau (Apr. 2003).

[20] J. Krinke, Context-sensitivity matters, but context does not, in: Proc. IEEE International Workshop on Source Code Analysis and Manipulation, 2004, pp. 29–35.

[21] B. G. Ryder, W. Landi, B. Philip, A. Stocks, S. Zhang, R. Altucher, A schema for interprocedural modification side-effect analysis with pointer aliasing, ACM Trans. Prog. Lang. Syst. 23 (2) (2001) 105–186.

[22] H. Agrawal, J. R. Horgan, Dynamic program slicing, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, 1990, pp. 246–256.

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